PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 9/46

A2

(11) International Publication Number: WO 00/10084

(43) International Publication Date: 24 February 2000 (24.02.00)

(21) International Application Number: PCT/US99/18813

(22) International Filing Date: 17 August 1999 (17.08.99)

(30) Priority Data:

09/135,106	17 August 1998 (17.08.98)	US
09/135,397	17 August 1998 (17.08.98)	US

(81) Designated States: JP, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published

Without international search report and to be republished upon receipt of that report.

(71) Applicant: MICROSOFT CORPORATION [US/US]; One Microsoft Way, Redmond, WA 98052 (US).

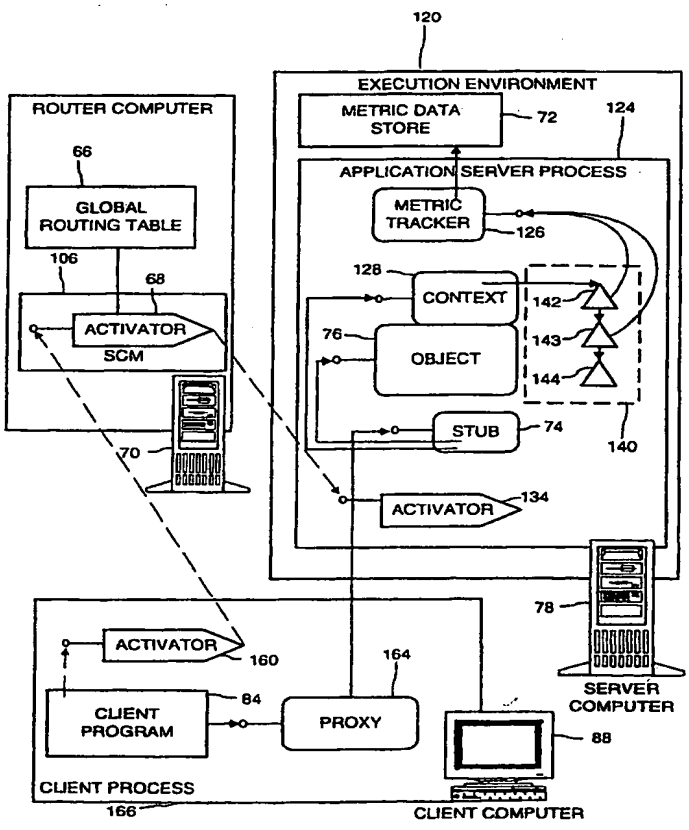
(72) Inventors: AL-GHOSEIN, Mohsen; 24677 S.E. 9th Place, Redmond, WA 98053 (US). GRAY, Jan, S.; 17214 N.E. 40th Street, Redmond, WA 98052 (US). MITAL, Amit; 13114 N.E. 111th Place, Kirkland, WA 98033 (US). LIMPRECHT, Rodney; 18806 N.E. 146th Way, Woodinville, WA 98072 (US).

(74) Agent: WIGHT, Stephen, A.; Klarquist, Sparkman, Campbell, Leigh & Whinston, LLP, One World Trade Center, Suite 1600, 121 S.W. Salmon Street, Portland, OR 97204 (US).

(54) Title: OBJECT LOAD BALANCING

(57) Abstract

A load balancing architecture routes object creation requests to server computers based on the object's class or some other characteristic. The architecture operates transparently to the object and the client program creating the object. A monitoring service at the servers observes a processing metric such as response time and provides the observations to a router computer. A load balancing engine at the router computer analyzes the observations to dynamically adjust routing, directing object creation requests to server computers having favorable performance. Analysis of observations is delayed to facilitate blending and dampening. Although a default load balancing engine is provided, software developers can specify a different engine per object class, and the engines can be tailored to the behavior of a particular object class. Load balancing is extensible to threads and processes. If performance at a server falls below an acceptable level, a rebalance policy transparently sends a rebalance message, breaking the connectionto the server.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

OBJECT LOAD BALANCING

TECHNICAL FIELD

The present invention relates to an object-based distributed computing model. More specifically, the invention relates to balancing an object-related resource load among a group of server computers.

BACKGROUND OF THE INVENTION

In many information processing applications, a server application running on a host or server computer in a distributed network provides processing services or functions for client applications running on terminal or workstation computers of the network which are operated by a multitude of users. Common examples of such server applications include software for processing class registrations at a university, travel reservations, money transfers and other services at a bank, and sales at a business. In these examples, the host or server computers access various resources to provide functionality to the client applications over the network. For example, a server computer might respond to a customer order from a client computer by consulting and updating price and inventory databases.

An important consideration in many of these applications is the ability to handle heavy processing loads. In the above application examples, for instance, the orders from a large number of users may be submitted to the server within a short time period. Processing each customer's order may consume significant processing resources. Thus, each additional user of the server application can slow the response or time to complete all other customers' orders, thus reducing the quality of service for all customers. Eventually, the load may exceed the processing capacity, possibly resulting in system failure, down time, and lost data. The degree to which a server can support incremental increases in user load while preserving performance is sometimes referred to as scalability.

One approach to enhancing scalability is to distribute processing among a group of server computers. For example, in a web server environment, a client computer with a web browser program requests web pages from a remote computer. To enhance scalability, various techniques can be used in a web server environment to accommodate a large number of requests in a short period. One approach is to send web page requests to a router, which directs the requests to a server computer designated to provide certain web pages. In this way, network traffic is divided among the server computers based on the web page requested. Another approach is to direct web page requests using a round robin or random server technique. This process of dividing processing among server computers is sometimes called load balancing.

Programming models generally known as object-oriented programming provide many benefits that have been shown to increase programmers' productivity, but the behavior of server

applications developed with object-oriented programming models makes it difficult to achieve enhanced scalability using the above-described load balancing techniques. In object-oriented programming, programs are written as a collection of object classes which each model real world or abstract items by combining data to represent the item's properties with functions to represent the item's functionality. More specifically, an object is an instance of a programmer-defined type referred to as a class, which exhibits the characteristics of data encapsulation, polymorphism and inheritance. Data encapsulation refers to the combining of data (also referred to as properties of an object) with methods that operate on the data (also referred to as member functions of an object) into a unitary software component (i.e., the object), such that the object hides its internal composition, structure and operation and exposes its functionality to client programs that utilize the object only through one or more interfaces. An interface of the object is a group of semantically related member functions of the object. In other words, the client programs do not access the object's data directly, but must instead call functions on the object's interfaces to operate on the data.

Polymorphism refers to the ability to view (i.e., interact with) two similar objects through a common interface, thereby eliminating the need to differentiate between two objects. Inheritance refers to the derivation of different classes of objects from a base class, where the derived classes inherit the properties and characteristics of the base class.

Object-oriented programming generally has advantages in ease of programming, extensibility, reuse of code, and integration of software from different vendors and (in some object-oriented programming models) across programming languages. However, when object-oriented programming techniques are used to develop server applications, the server applications generally exhibit behavior not accommodated by the above-described load balancing techniques. For example, object-oriented applications typically create software objects on a server, and the software objects themselves may use resources or require creation of additional software objects while they reside on the server. By contrast, the above techniques assume that an incoming request is for a particular single resource, such as a web page. In addition, a software object may require its state be preserved between method calls. Accordingly, method calls from the client should be directed to the server on which the object was created so that the object's preserved internal state data can be readily accessed. By contrast, the above techniques assume that an incoming request can be routed to any server without regard to prior requests. As a result, the above-described load balancing techniques fail to address behavior characteristic of software objects and offer limited scalability to server applications employing software objects.

SUMMARY OF THE INVENTION

The present invention is an extensible object-based load balancing architecture for use with applications created using object-oriented programming techniques. A load balancing service in the

architecture supports dynamic object-based load balancing among a plurality of servers to enhance server application scalability. Objects in the architecture need not have load balancing logic because load balancing logic is incorporated into the architecture. In other words, load balancing is achieved transparently to objects, and software developers need not tailor objects to the load balancing architecture. As a result, the architecture accommodates a wide variety of software objects, and application software developers can develop objects without regard for the load balancing arrangement. Software developers are freed to focus on incorporating business logic into objects and are relieved from developing their own load balancing architecture.

In one aspect of the invention, object creation requests are sent to a router computer, which sends the request to an appropriate server computer based on the object class supplied with the request. In this way, objects of an object class known to have particular resource requirements or certain patterns of behavior can be routed to a server best equipped to handle creating and hosting the objects.

In another aspect of the invention, dynamic load balancing is achieved by collecting information about a processing metric at the server computers and forwarding the information to a load balancing engine on the router computer. The load balancing engine modifies how the router computer divides object creation requests among the server computers. A default load balancing engine for monitoring response time is available, but system software developers can construct custom load balancing engines for particular applications. Processing metric information is collected transparently to the object and the client program using the object, again freeing application software developers from including load balancing logic in the application's objects.

In addition, a different load balancing engine can be specified for different object classes; thus, the load balancing engine can be tailored to specific object idiosyncrasies. Since the load balancing logic and the business logic are separated, developers can design load balancing engines independently of objects, and the load balancing logic or parameters can be modified after application deployment without affecting the business logic. Creating a load balancing service typically requires intimate knowledge of technical detail associated with communication networks and computer hardware. The architecture's arrangement has the further benefit of allowing one developer group with application development expertise to focus on the business logic of the server application, while another group with appropriate technical expertise develops the load balancing logic or associated load balancing engines. In this way, the benefits of load balancing can be more easily incorporated into both existing and future application software. Such an arrangement particularly facilitates the advancement of electronic commerce applications, which commonly require enhanced scalability and business logic that evolves over time.

In another aspect of the invention, the logic for selecting a server computer to host an object creation request is loosely coupled with the logic observing processing metrics to provide

dampening of variance typical in processing metric data. As a result, load balancing engines avoid considering anomalous observations.

In another aspect of the invention, a rebalance arrangement transparently sends a notification to a client computer using an object on a server computer. The notification breaks the client-server connection and an object creation request is resubmitted. In this way, the server computer load can be rebalanced transparently, and the load balance architecture accommodates recycled objects and just-in-time activation.

In yet another aspect of the invention, servers are placed into a set called a target group. Target groups are extensible to encompass alternative resources, such as threads and processes.

Additional features and advantages of the invention will be made apparent from the following detailed description of an illustrated embodiment, which proceeds with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram of a computer system that may be used to implement a method and apparatus embodying the invention for balancing a load of object creation requests among a group of servers.

Fig. 2 is block diagram showing a broad overview of an object-based load balancing architecture, including a router, server, and client computers such as those shown in Fig. 1.

Fig. 3 is a block diagram of a computer network showing a router computer receiving a client computer object creation request and routing the request to a server computer selected from a global routing table.

Fig. 4 is a block diagram of an object conforming to the Common Object Model specification of Microsoft Corporation.

Fig. 5 is a flowchart showing a method for creating an instance of a software object on a network such as that shown in Fig. 3.

Fig. 6 is a flowchart showing a method for invoking a method call to a software object, such as that created by the method shown in Fig. 5.

Fig. 7 is a block diagram showing a load balancing service on a router computer collecting information from a server computer over a network.

Fig. 8 is a block diagram showing a computer network with a client and server computer implementing a retry object policy for generating a rebalance message transparently to a server object and a client program.

Fig. 9 is a flowchart showing a method for implementing a rebalance message over a computer network such as that shown in Fig. 2.

Fig. 10 is a view showing a user interface for configuring a load balance service such as that shown in Fig. 7.

Figure 11 is a block diagram of an extensible object execution environment running on a distributed network of computers such as that shown in Figure 1, in which component application objects (such as shown in Figure 4) at an intersection of domains are associated in contexts.

Figure 12 is a block diagram of an object context object that represents a context in the extensible object execution environment of Figure 11.

Figure 13 is a program listing defining an "IObjectContext" interface of the object context object of Figure 12.

Figure 14 is a block diagram of a cross-context reference and policy set in the extensible object execution environment of Figure 11 for the same apartment case.

Figure 15 is a block diagram of a cross-context reference and policy set in the extensible object execution environment of Figure 11 for the cross-apartment, cross-process, and/or cross-machine case.

Figure 16 is a program listing defining an "IPolicy" interface of a policy object in the cross-context references of Figures 14 and 15.

Figure 17 is a block diagram of a policy set object that represents the policy set in the cross-context references of Figures 14 and 15.

Figure 18 is a program listing defining an "IPolicySet" interface of the policy set object of Figure 17.

Figure 19 is a program listing defining an "IPolicyMaker" interface of policy makers in the object context object of Figure 12.

Figure 20 is a block diagram of cross-context reference tracking data structures in the object execution environment of Figure 11.

Figure 21 is a block diagram of data structures implementing the same-apartment, cross-context reference of Figure 14.

Figure 22 is a block diagram of data structures implementing a client side of the cross-apartment or -process, cross-context reference of Figure 15.

Figure 23 is a block diagram of data structures implementing a server side of the cross-apartment or -process, cross-context reference of Figure 15.

Figure 24 is a program listing defining an "IWrapper" interface of a wrapper object in the same apartment, cross-context reference implementing data structures of Figure 21.

Figure 25 is a program listing defining an "ICtxChannel" interface of a context channel object in the same apartment, cross-context reference implementing data structures of Figure 21.

Figure 26 is a block diagram of a chain of activators that establish a context of a component application object in the object execution environment of Figure 11.

Figure 27 is a flow diagram of activation data in the chain of activators of Figure 26.

Figures 28 and 29 are program listings defining interfaces used in the activation chain of Figure 26.

DETAILED DESCRIPTION OF THE INVENTION

The present invention is directed toward a method and system for object-based load balancing. In one embodiment illustrated herein, the invention is incorporated into an object services component, entitled "COM+," of an operating system, entitled "Microsoft Windows NT Server 5.0," marketed by Microsoft Corporation of Redmond, Washington. Briefly described, this software is a scaleable, high-performance network and computer operating system supporting distributed client/server computing, and providing an object execution environment for object applications conforming to COM. The COM+ component incorporates object services from prior object systems, including Microsoft Component Object Model (COM), Microsoft Object Linking and Embedding (OLE), Microsoft Distributed Component Object Model (DCOM), and Microsoft Transaction Server (MTS).

Exemplary Operating Environment

Figure 1 and the following discussion are intended to provide a brief, general description of a suitable computing environment in which the invention may be implemented. While the invention will be described in the general context of computer-executable instructions of a computer program that runs on a computer, those skilled in the art will recognize that the invention also may be implemented in combination with other program modules. Generally, program modules include routines, programs, components, objects, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, minicomputers, mainframe computers, and the like. The illustrated embodiment of the invention also is practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. But, some embodiments of the invention can be practiced on stand alone computers. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

With reference to Figure 1, an exemplary system for implementing the invention includes a conventional computer 20 (e.g., a server computer, a personal computer or other like computer), including a processing unit 21, a system memory 22, and a system bus 23 that couples various system components including the system memory to the processing unit 21. The processing unit may be any of various commercially available processors, including Intel x86, Pentium and compatible microprocessors from Intel and others, including Cyrix and AMD; Alpha from Digital; MIPS from MIPS Technology, NEC, IDT, Siemens, and others; and the PowerPC from IBM and

Motorola. Dual microprocessors and other multi-processor architectures also can be used as the processing unit 21.

The system bus may be any of several types of bus structure including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of conventional bus architectures such as PCI, VESA, Microchannel, ISA and EISA, to name a few. The system memory includes read only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system (BIOS), containing the basic routines that help to transfer information between elements within the computer 20, such as during start-up, is stored in ROM 24.

The computer 20 further includes a hard disk drive 27, a magnetic disk drive 28, e.g., to read from or write to a removable disk 29, and an optical disk drive 30, e.g., for reading a CD-ROM disk 31 or to read from or write to other optical media. The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of data, data structures, computer-executable instructions, etc. for the computer 20. Although the description of computer-readable media above refers to a hard disk, a removable magnetic disk and a CD, it should be appreciated by those skilled in the art that other types of media which are readable by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, and the like, may also be used in the exemplary operating environment.

A number of program modules may be stored in the drives and RAM 25, including an operating system 35, one or more application programs 36, other program modules 37, and program data 38.

A user may enter commands and information into the computer 20 through a keyboard 40 and pointing device, such as a mouse 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port or a universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor, computers typically include other peripheral output devices (not shown), such as speakers and printers.

The computer 20 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 49. The remote computer 49 may be a server, a router, a peer device or other common network node, and typically includes many or all of the elements described relative to the computer 20, although only a memory storage device 50 has been illustrated in Figure 1. The logical connections depicted in Figure 1 include a local area network (LAN) 51 and a wide area network (WAN) 52. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN networking environment, the computer 20 is connected to the local network 51 through a network interface or adapter 53. When used in a WAN networking environment, the computer 20 typically includes a modem 54 or other means for establishing communications over the wide area network 52, such as the Internet. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the computer 20, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

In accordance with the practices of persons skilled in the art of computer programming, the present invention is described below with reference to acts and symbolic representations of operations that are performed by the computer 20, unless indicated otherwise. Such acts and operations are sometimes referred to as being computer-executed. It will be appreciated that the acts and symbolically represented operations include the manipulation by the processing unit 21 of electrical signals representing data bits which causes a resulting transformation or reduction of the electrical signal representation, and the maintenance of data bits at memory locations in the memory system (including the system memory 22, hard drive 27, floppy disks 29, and CD-ROM 31) to thereby reconfigure or otherwise alter the computer system's operation, as well as other processing of signals. The memory locations where data bits are maintained are physical locations that have particular electrical, magnetic, or optical properties corresponding to the data bits.

Architecture Overview

Figure 2 presents a broad overview of the load balancing architecture comprising computers such as the computer 20 (Fig. 1). A group of client computers 82 sends object creation requests to a router computer 70, which selects one of the server computers in the server group 80 to host the request. The illustrated computers are connected by a network, such as the Internet or an office LAN or WAN, over which the requests are sent.

A client program 84 running on the client computer 88 issues a request to create an object of a particular class (typically, an object of a server application that runs on server computers of the network). The request is directed to an activator 68 on the router computer 70, which selects an appropriate server computer from the server group 80 to process the object creation based on information in a global routing table 66. Typically, the global routing table 66 associates the class of the object with a particular server or servers. Then, the server object 76 and an associated stub 74 are created on the selected server computer. As described in more detail below, the stub 74 can monitor calls to the object 76 and is therefore sometimes called an object wrapper. The stub 74 forwards object references to the server object 76 and also observes and records a processing metric (e.g., response time), which is sent to a metric data store 72 with the

assistance of other objects as described in more detail below. Information in the metric data store 72 is sent to the router computer 70 and used by a load balancing engine 62 to modify the global routing table 66. Thus, subsequent object creation requests are routed by the activator 68 according to the modified information in the global routing table 66. As requests from other client computers are routed to other server computers in the group 80, the other server computers similarly collect information in their own metric data stores, which is sent to the router computer 70 for use by the load balancing engine 62.

A default load balancing engine is provided, but the load balancing architecture can accommodate a different load balancing engine for use with objects of different classes. Thus, the load balancing engine can analyze observations with logic customized according to an object's particular behavior. Accordingly, the system can accommodate an additional load balancing engine 64 for processing data related to processing metric observations for another class of objects.

Since information is collected from the server group 80 on an ongoing basis, the router computer 70 routes object creation requests dynamically. For example, several requests might be sent to a first server computer until the load balancing engine 62 determines that the response time of the server has dropped below that of a second server computer. The load balancing engine 62 could then inform the activator 68 that objects should be created on the second server machine by modifying an entry in the global routing table 68. Thus, requests for the same class of objects might be sent to a different machine at different times, depending on observed processing metrics such as response time.

Server Application Execution Environment

With reference now to Figure 3, the operating system of the server computer 78 provides run-time or system services to create a run-time execution environment 120 that supports control of a server object (e.g., the server object 76) over its state duration on the server computer 78. The operating system also provides services for thread and context management to the server object 76. The execution environment 120 is provided by a modified version of Microsoft Transaction Server, marketed by Microsoft Corporation. Microsoft Transaction Server is a well-known, widely available off-the-shelf software product providing an execution environment for software objects implementing transactions. In the illustrated embodiment, Microsoft Transaction Server is integrated in the operating system as part of the services provided by COM+.

The illustrated application server process ("ASP") 124 is a system process that hosts execution of server objects. Each ASP 124 can host multiple server objects that are grouped into a collection called an "application." Also, multiple ASPs 124 can execute on the server computer 78 under a multi-threaded, multi-tasking operating system (e.g., Microsoft Windows NT in the illustrated embodiment). Each ASP 124 provides a separate trust boundary and fault isolation domain for the server objects. In other words, when run in separate ASPs, a fault by one server object which causes its ASP to terminate generally does not affect the server objects in another

ASP. In the illustrated embodiment, server objects are grouped as an application to be run together in one ASP 124 using an administration utility providing a graphical user interface for managing attributes associated with server objects, including grouping the objects into applications.

In a typical installation shown in Figure 3, the execution environment 120 is on the server computer 78 (which may be an example of the computer 20 described above) that is connected in a distributed computer network comprising a large number of client computers 88 which access the server objects in the execution environment 120. Alternatively, the execution environment 120 may reside on a single computer and host server objects accessed by client processes also resident on that computer.

Server Objects

The server objects 76 hosted in the execution environment 120 of the ASP 124 implement the business logic of a server application, such as the code to manage class registrations in a university's registration application or orders in an on-line sales application. Typically, each server application comprises multiple objects, each of which contains program code for a portion of the application's work. For example, a banking application may comprise a transfer object, a debit account object, and a credit account object, which perform parts of the work of a money transfer operation in the application.

With reference now to Figure 4, the server object 76 (Figure 3) in the illustrated embodiment conforms to the Component Object Model ("COM") of Microsoft Corporation's OLE and ActiveX specifications (i.e., is implemented as a "COM Object"), but alternatively may be implemented according to other object standards including the CORBA (Common Object Request Broker Architecture) specification of the Object Management Group. OLE's COM specification defines binary standards for components and their interfaces, which facilitate the integration of software components. For a detailed discussion of OLE, see Kraig Brockschmidt, Inside OLE, Second Edition, Microsoft Press, Redmond, Washington, 1995.

In accordance with COM, the server object 76 is represented in the computer system 20 (Fig. 1) by an instance data structure 202, a virtual function table 204, and member functions 206-208. The instance data structure 202 contains a pointer 210 to the virtual function table 204 and data 212 (also referred to as data members, or properties of the object). A pointer is a data value that holds the address of an item in memory. The virtual function table 204 contains entries 216-218 for the member functions 206-208. Each of the entries 216-218 contains a reference to the code 206-208 that implements the corresponding member function.

The pointer 210, the virtual function table 204, and the member functions 206-208 implement an interface of the server object 76. By convention, the interfaces of a COM object are illustrated graphically as a plug-in jack as shown for the server object 76 in Fig. 3. Also, interfaces conventionally are given names beginning with a capital "I." In accordance with COM, the server object 76 can include multiple interfaces, which are implemented with one or more

virtual function tables. The member function of an interface is denoted as "IInterfaceName::FunctionName."

The virtual function table 204 and member functions 206-208 of the server object 76 are provided by a server application program 220 (hereafter "server application DLL"), which is stored in the server computer 78 (Figure 3) as a dynamic link library file (denoted with a ".dll" file name extension). In accordance with COM, the server application DLL 220 includes code for the virtual function table 204 and member functions 206-208 of the classes that it supports, and also includes a class factory 222 that generates the instance data structure 202 for an object of the class.

Like any COM object, the sever application object can maintain internal state (i.e., its instance data structure 202 including data members 212) across multiple interactions with a client (i.e., multiple client program calls to member functions of the object). The server object that has this behavior is said to be "stateful." The server object can also be "stateless," which means the object does not hold any intermediate state while waiting for the next call from a client.

Returning now to Fig. 3, before the server object 76 can execute in the illustrated execution environment 120, the server object 76 is first installed on the server computer 78. A typical object installation involves installing a group of related objects called an "application" (also known as a "package"). As with any COM object, the server object 76 is installed by storing the server application DLL file 220 (Fig. 4) that provides the server object 76 in data storage accessible by the server computer (typically the hard drive 27, shown in Figure 1, of the server computer), and registering COM attributes (e.g., class identifier, path and name of the server application DLL file 220 (Fig. 4), etc. as described below) of the server objects in the system registry. The system registry is a configuration database.

In the execution environment 120 of Figure 3, the server object 76 is executed under control of the operating system in the ASP 124. The operating system is responsible for loading the server application DLL 220 (Fig. 4) into the ASP 124 and instantiating the server object 76 using the class factory 222 (Fig. 4). The operating system further manages calls to the server object 76 from client programs (whether resident on the same computer or over a network connection).

The illustrated execution environment 120 imposes certain additional requirements on the server object 76 beyond conforming with COM requirements. First, the server object is implemented in a DLL file (i.e., the server application DLL 220 of Figure 4). (COM objects otherwise alternatively can be implemented in an executable (".exe") file.) Second, the object's DLL file 220 has a standard class factory 222 (i.e., the DLL implements and exports the DllGetClassObject function, and supports the IClassFactory interface). Third, the server object exports only interfaces that can be standard marshaled, meaning the object's interfaces are either described by a type library or have a proxy-stub DLL. The proxy-stub DLL provides a proxy 164 in a client process 166 on the client computer 88, and a stub 74 in the ASP 124 on the server

computer 78. The proxy 164 and stub 74 marshal calls from a client program 84 across to the server computer 78. These additional requirements conform to well known practices.

The client program 84 is a program that accesses the functionality of server object 76. The client program 84 can be program code (e.g., an application program, COM Object, etc.) that runs outside of an ASP. Such client programs are referred to as "base clients." Alternatively, the client program 84 can be another server object that also runs in the same or a separate ASP. The client program 84 can reside on the server computer 78 or on a separate client computer 88 as shown in Figure 3.

Overview Of COM Object Instantiation In OLE

As with other COM objects, the client program 84 (Figure 3) must first request creation of an instance of the server object 76 and obtain a reference to the server object 76 before the client program 84 can access the functionality implemented by the server object 76 (i.e., before the client program can call member functions supported on an interface of the server object).

In Microsoft's OLE, a client program instantiates a COM object using services provided by OLE and a set of standard object interfaces defined by COM based on class and interface identifiers assigned to the object's class and interfaces. More specifically, the services are available to client programs as application programming interface (API) functions provided in the COM+ library, which is part of a component of the Microsoft Windows operating system in a file named "OLE32.DLL." Also, in OLE, classes of COM objects are uniquely associated with class identifiers ("CLSIDs"), and registered by their CLSID in a system configuration database referred to as the "registry." The registry entry for a COM object class associates the CLSID of the class with information identifying an executable file that provides the class (e.g., a DLL file having a class factory to produce an instance of the class). Class identifiers are 128-bit globally unique identifiers ("GUIDs") that the programmer creates with an OLE service named "CoCreateGUID" (or any of several other APIs and utilities that are used to create universally unique identifiers) and assigns to the respective classes. The interfaces of an object additionally are associated with interface identifiers ("IIDs").

In particular, the COM+ library provides an API function, "CoCreateInstance," that the client program can call to request creation of an object using its assigned CLSID and an IID of a desired interface. In response, the CoCreateInstance API looks up the registry entry of the requested CLSID in the registry to identify the executable file for the class. The CoCreateInstance API function then loads the class' executable file, and uses the class factory in the executable file to create an instance of the COM object. Finally, the CoCreateInstance API function returns a pointer of the requested interface to the client program. The CoCreateInstance API function can load the executable file either in the client program's process, or into a server process which can be either local or remote (i.e., on the same computer or a remote computer in a distributed computer network) depending on the attributes registered for the COM object in the system registry.

Once the client program has obtained this first interface pointer of the COM object, the client can obtain pointers of other desired interfaces of the object using the interface identifier associated with the desired interface. COM defines several standard interfaces generally supported by COM objects including the IUnknown interface. This interface includes a member function named "QueryInterface." The QueryInterface function can be called with an interface identifier as an argument, and returns a pointer to the interface associated with that interface identifier. The IUnknown interface of each COM object also includes member functions, AddRef and Release, for maintaining a count of client programs holding a reference (e.g., an interface pointer) to the COM object. By convention, the IUnknown interface's member functions are included as part of each interface on a COM object. Thus, any interface pointer that the client obtains to an interface of the COM object can be used to call the QueryInterface function.

Remote Procedure Call Facility

To achieve distributed computing across network connections, the illustrated computers implement a remoting facility. The Microsoft Distributed Component Object Model (DCOM) provides a remote procedure call (RPC) remoting facility (hereafter "DCOM RPC remoting facility") that allows transparent interface function calls across process and machine (i.e., computer) boundaries. (See, e.g., Brockschmidt, Inside OLE, Second Edition 277-338 (1995).)

For transparency, the DCOM RPC remoting facility provides marshaling code (referred to as a "proxy") inside the process of a client program, component or object (the "client") that is making an interface call to an out-of-process or remote-computer-resident object (the "server object"), and also provides unmarshaling code (referred to as a "stub") in the process of the server object. The proxy receives the client's in-process interface call and marshals all data needed for the call (e.g., arguments and in-memory data) into a buffer for transfer to the stub over a communications channel (the "RPC channel") between the client and server processes or machines. The stub unmarshals the data from the buffer at the server object's process and machine and directly invokes the interface call on the server object. The stub also marshals any return value and "out" parameters returned from the interface call for transfer to and unmarshaling by the proxy for passing on to the client.

This remote procedure calling is transparent to the client and server object in that the DCOM RPC remoting facility automatically provides the proxy, the stub and the RPC channel for marshaling the interface call across process and machine boundaries, such that the client and server object can perform the interface call as if both are on the same computer and in the same process. In the illustrated embodiments, the DCOM RPC remoting facility and various other object-related services related to distributed computing are provided by a set of COM+ libraries available in the operating system called the service control manager (or SCM).

Creating an Object in the Load Balancing Architecture

With reference now to Figure 3, a client program 84 requiring access to server object functionality runs in a client process 166 on a client computer 88. To request a local object, the client program 84 calls the CoCreateInstance API function described above. If an object is to be created remotely, information in the registry is ordinarily set to indicate that object instantiation requests for objects of a particular class are to be sent to an appropriate server. In the case of a load balanced object, the registry indicates requests are to be sent to the router computer 70, and a call to the CoCreateInstance API results in the steps shown in Figure 5.

In addition to the standard COM functions described above, the COM+ component of the operating system in the illustrated exemplary embodiment provides services relating to contexts, policies and activators, which are further described in other sections, below. Specifically, the load balancing architecture of Figure 3 employs activators to implement object creation requests. An activator participates in administrative tasks related to the object creation process and in the exemplary illustrated embodiment is a software object implementing the IActivator interface. Activators are automatically invoked by the COM+ infrastructure upon an API call to CoCreateInstance. An activator's tasks can include determining the computer on which an object should be created and providing references to other objects assembled or created at object creation time. Conceptually, an activator creates a wrapper around an object with a stub as explained in more detail below. When an activator receives an object creation request, the activator expresses the request as activation properties indicating the class identifier of the object, an identifier indicating on what machine the object should be created and other related information.

Activators typically work together in a chain by successively manipulating the activation properties and passing the properties to a next activator in the chain. In the exemplary illustrated embodiment, the process of activation (sometimes called an activation delegation chain) goes through various stages in a particular order. When an activator is finished with its particular tasks, it delegates activation to the next stage. The COM+ infrastructure controls which activator is invoked when an activator delegates onward. As each activator is called, it may create or locate other objects and provide object references. Typically, the last activator in a chain (sometimes called a "base" activator) calls a class factory to create a new instance of the requested object, but base activators may take other actions, such as recycling an object as explained more fully below. Activators can run in various environments, such as in a service control manager, in a client process, or in an ASP. On each computer, the registry is set to associate each object CLSID with appropriate activators for each stage; a default activator is used if an activator for a particular stage is not specified. A chain of activators can include activators running in plural processes on a given computer, and the chain can span plural computers.

An activator can direct activation to a different computer by setting the server property of the activation properties to the desired computer and delegating to the next stage. When the

activation chain crosses a machine boundary, the typical activation delegation chain includes a client-side SCM stage to run an activator in the client's SCM and a subsequent server-side SCM stage to run an activator in the server's SCM; however, some agent other than the SCM could host the load balancing activator. In the illustrated embodiment, the router computer 70 directs an object creation request to an appropriate server by appropriately setting the activation properties in the SCM stage.

Figure 5 illustrates a method for creating a server object in the load balancing architecture shown in Fig. 3. At step 302, the client program's creation request is received by the client activator 160 (Fig. 3). In the illustrated example, the requested object is a load balanced object, and the activation properties derived from the registry indicate the creation request is to be directed to the router computer 70. At step 304 (Fig. 5), the client activator 160 sends the client program's object creation request to the router computer 70 using the DCOM RPC remoting facility. To facilitate transmission of the request, the activator 160 uses services of the client computer's service control manager (not shown), which may involve other activators.

At the router computer 70, a global routing table 66 associates class identifiers with server identifiers in a table structure. In the illustrated embodiment, a hash table is used to associate a class identifier with a server identifier, but some other data structure could alternatively be used. Multiple class identifiers may be associated with the same server. Typically, a class identifier is associated with a single server computer; however, an alternative global routing table 66 could associate a class identifier to multiple servers. In addition to the typical COM attributes, the CLSIDs of objects to be load balanced among the server computers are registered in the system registry (or alternatively in a configuration database called the "catalog") with a "load balance" attribute indicating that the object is to be load balanced. Additional information indicates a target group for the CLSID, as is more fully explained below.

At step 306 (Fig. 5), when the router activator 68 receives the object creation request, the router activator 68 determines if the object is to be load balanced and consults the global routing table 66 to determine the appropriate server computer 78 to host the creation request according to the class identifier in the activation properties. If multiple servers are associated with the class identifier, a random or round robin algorithm can be used to select a server. In this way, requests to create objects of a particular class are routed to a particular server or set of servers. Preferably, the router activator 68 responds quickly. As will be explained more fully below, quick response is achieved in the illustrated embodiment because load balancing engines, not the activator 68, generally carry the burden of periodically updating the global routing table 66. The router activator 68 need only consult the global routing table 66, which provides accelerated look up by associating a hash function on the class identifier with the appropriate server. In this way, a server can be determined quickly from a class identifier. The router activator 68 sets the server identifier property of the activation properties to the selected server identifier, which causes the creation

request to be forwarded by the COM+infrastructure to the server activator 134 at the server identified by the server identifier. Forwarding the request is also sometimes called "delegating" the request to a server. In the illustrated example, the router activator 68 runs in the router's service control manager (SCM) 106, which provides additional services related to the DCOM RPC remoting facility.

In an alternative embodiment, instead of consulting the global routing table 66, the router activator 68 consults some other host server selection means to determine to which server the creation request should be delegated. For example, the router activator 68 can be configured to consult a load balancing engine 62 method comprising code to select an appropriate host server (e.g., a server having a superior performance metric or a server known to be hosting like objects) based on a specified characteristic of the object creation request (e.g., class identifier, client computer identity, client process identity or database requirements). The load balancing engine supplies the selected host server to the activator 68, which delegates the creation request to the selected host server.

The router activator 68 can also be configured to maintain a list of past object creation requests, including the requesting process, the requesting client computer, and the requested class identifier. In this way, the router activator 68 can be configured to route subsequent requests by a process to the same server computer used for the process' past requests. As a result, the client process 166 avoids creating objects on plural server computers, which requires additional connections and may result in degraded performance. Also, the router activator 68 can be configured to route requests by a client computer for objects of the same class identifier to the server used for the client's past requests to provide client affinity.

The server computer 78 also includes an SCM (not shown) which fields and forwards the request to the server activator 134; the SCM may itself contain other activators. At step 308 (Fig. 5), the server activator 134 receives the object creation request. The server activator 134 creates the server object 76 and performs administrative tasks related to the object creation request, including arranging for creation of an object context object 128, an object policy set 140 and a stub 74. The server object 76 and associated elements 74, 128, and 140 are sometimes called an "object cluster;" each of the elements of the object cluster can be implemented as an object (e.g., the stub 74 can be implemented as an object). In effect, the server activator 134 creates a wrapper around the server object 76; subsequent calls to the server object 76 are first routed to the stub 74, which can consult the object context object 128 and object policy set 140. The wrapper, however, is transparent to the client program 84 and the server object 76 in that neither needs to incorporate logic for creating or maintaining the wrapper.

At step 310 (Fig. 5), the server activator 82 creates an object cluster comprising the server object 76, a stub 74 and an object context object 128 with associated object context properties. An appropriate class factory (in a server application DLL, as explained above) creates the server object

76 in an ASP 124 which runs in an execution environment of the selected server computer 78. The stub 74 is responsible for unmarshalling calls to the server object 76 and additionally maintains a reference to the object context object 128 associated with the server object 76. The object context object 128 provides context for the execution of the server object 76 in the execution environment 120. The object context object 128 has a lifetime that is coextensive with that of the server object 76. The server activator 134 creates the object context object 128 when the server object 76 is initially created, and the object context object 128 is destroyed after the server object 76 is destroyed (i.e., after the last reference to the server object is released).

The object context object 128 maintains object context property objects of the server object 76; the object context property objects are determined by the server activator 134 (and possibly other activators) when the server object 76 is created and define the server object's context. Generally, if two objects share the same set of object context property objects, they are said to be in the same context. When a reference to the object 76 is unmarshaled, the object context property objects are consulted to provide certain functions 140 to be run automatically when actions are performed on the server object 76. These certain functions are called object policies 142-144; in the illustrated embodiment, the functions are IPolicy interfaces on objects, which may be the context property objects themselves.

The set of object policies for a particular object is called the object's policy set 140. Thus, the activator can arrange for certain policies to be included in an object's policy set by including a context property object providing the appropriate policy. A context property object providing policies is sometimes called a policy maker because it contributes policies to the policy set; in the illustrated embodiment, a context property object implementing the IPolicyMaker interface is a policy maker. In the illustrated exemplary embodiment, the context property object, policy maker and policy may be implemented as a single object with appropriate interfaces for each of its functions. Alternatively, separate objects could be used. Thus, in the illustrated embodiment, an object's context is defined as an object environment provided by the policy set associated with the object. Context could alternatively be provided by some other mechanism (e.g., code in the RPC or stub layer).

Particular policies 142-144 are invoked according to the action performed on the server object 76. For example, policies called call and return policies typically run on the proxy (e.g., client) side before method calls are passed to the object and before the method returns, respectively. Policies called enter and leave policies typically run on the stub (e.g., server) side immediately before a call and immediately before a return, respectively. As a result, the following is typical of object policy execution order for a method call to an object with a policy set:

1. Issue method call from client program to proxy
2. Execute call policies
3. Direct call to stub
4. Execute enter policies

5. Execute object method, return results to stub
6. Execute leave policies
7. Return method results to proxy
8. Execute return policies
9. Return method results to client program

Additionally, an object policy may communicate with other object policies using a buffer mechanism. In the illustrated embodiment, object policies are implemented as software objects having an IPolicy interface, which is defined as follows:

```
Interface IPolicy : IUnknown {
      Call(pcall), Enter(pcall), Leave(pcall), Return (pcall);
      CallGetSize(pcall, pcb),
      CallFillBuffer(pcall,pcb,pvBuf),
      EnterWithBuffer(pcall,cb,pvBuf);
      LeaveGetSize(pcall,pcb),
        LeaveFillBuffer(pcall,pcb,pvBuf),
        ReturnWithBuffer(pcall,cb,pvBuf);
      ReleasePolicy(pcall);
      Signal(ceType,pcall);
};
```

Alternatively, additional methods can be implemented in the interface, such as a method called when the object is created and a method called when the object is disposed of or released.

In the case of a load balanced object in the illustrated embodiment, at least one of the context property objects added by the activator 134 is a policy maker contributing a load balancing policy to the object context object associated with the load balanced object. Typical load balancing policies include a metric tracker creator 142 and processing (or performance) metric measurers 143-144. In the illustrated example, the load balance object policies 142-144 facilitate measuring response time of method calls to the server object 76. However, as explained below, alternative metrics can be used, and the various illustrated functions can be implemented using fewer policies.

The object policy 142 creates an instance of a metric tracker object 126 for tracking response time if none already exists within the ASP 124. A default metric tracker 126 tracks response time; however, other metric trackers may be incorporated into the architecture for tracking other processing metrics. The type of metric tracker 126 is determined with reference to the class identifier of the server object 76. The determination can be made either at the server computer 78 or at the router computer 70, to facilitate central control. If control is exercised at the router computer 70, the router activator 68 sets an appropriate activation property during object instantiation time to indicate the desired metric tracker. A reference to the metric tracker 126 is provided to enter policy 143 and leave policy 144. The enter policy 143 runs before method calls and the leave policy 144 runs after method calls to the server object 76 to track response time using the metric tracker 126 as more fully explained below.

The object context object 128 may also contain transaction related information for the server object 76, such as a client id, an activity id, and a transaction reference. The client id refers

to the client program 84 that initiated creation of the server object 76. The activity id refers to an activity that includes the server object 76. An activity is a set of objects executing on behalf of a base client, within which only a single logical thread of execution is allowed. The transaction reference indicates a transaction property object (not shown) that represents a transaction in which the server object 76 participates. The object context object 128 is implemented as a COM object that runs under control of the stub 74.

In the illustrated execution environment, the stub 74 maintains an implicit association of the object context object 128 to the server object 76. In other words, the stub 74 does not pass a reference of the object context object 128 to the client program 84 using the server object 76. Rather, the stub 74 maintains the object's association with the context object 128, and accesses the object context object 128 when needed during the client program's access to the server object 76. Thus, the client program 84 is freed from explicitly referencing the object context object 128 while creating and using the server object 76. Finally, an object cluster may be created using just-in-time activation without actually creating a server object 76 as described more fully below.

At step 312 (Fig. 5), the server activator 134 provides a reference to the stub 74 back to the client activator 160, which creates a proxy 164. In the exemplary illustrated embodiment, the call returns first to the router activator 68, then back to the client activator 160. At step 314 (Fig. 5), the stub reference is provided to the proxy 164, and a reference to the proxy 164 is provided to the client program 84, completing the connection from the client program 84 to the server object 76. The client program 84 can thus access the functionality of the server object 76 through the proxy 164 and the stub 74, without need for involvement of the router computer 70. Typically, the reference provided to the proxy 164 is a pointer to an interface of the server object 76, through which the client program 84 can request additional interfaces and access the functionality of the server object 76.

Invoking an Object Method

Still with reference to Figure 3, when the client program 84 requires access to the functionality of the server object 76, the client program 84 invokes (or calls) a method of the server object 76, resulting in the steps illustrated in Figure 6.

Figure 6 shows a method for tracking a processing metric transparently to the client program 84 (Fig. 3) and the server object 76. At step 402 (Fig. 6), the client program 84 sends a method call to the proxy 164, which marshals the call for transmission to the server stub 74 via the DCOM RPC remoting facility using the interface reference provided as explained above. At step 404, the sever stub 74 receives the call.

Upon receiving the method call, the server stub 74 unmarshals the call and consults the object context object 128 to identify which object policies 140 are to be executed before the method call (the enter policies). The load balanced object cluster includes an enter policy 143 that works in conjunction with the metric tracker 126 to record a value indicative of the time of the call, using a

real time clock or number of processor ticks. The enter load balance policy is executed at step 406. After any other appropriate object policies are executed, at step 408, the stub 74 passes the method call to the server object, which executes the designated method using accompanying parameters, if any.

When the method call returns to the stub 74, the stub 74 again consults the object context object 128 to identify object policies to be run before the stub 74 returns the method call to the proxy 164 (the leave policies). The load balanced object cluster includes a leave policy 144 that works in conjunction with the metric tracker 126 to determine and record the response time of the method call using the time value recorded by the enter policy 143 as described above. The leave load balance policy 144 executes at step 410 and determines response time by subtracting the recorded time value before the call from the time value before the return (after the call). At step 412, the results of the method call are returned to the client program 84 using the DCOM RPC remoting facility. The metric tracker 126 eventually passes the information about the response time to the metric data store 72 at step 414. In this way, a processing metric is observed and recorded transparently to the server object 76 and the client program 84.

The execution environment of the server accommodates multiple ASPs 124, and each ASP 124 may include plural server objects 76, which may be of various object classes. A metric tracker 126 runs in each ASP 124 having load balanced objects, and each object class may be associated with a separate metric tracker or share a metric tracker. Thus, multiple metric trackers 126 can run in the server's execution environment; the metric trackers share the metric data store 72.

Over the course of time as various method calls are made, observations of response time by the metric tracker 126 are accumulated in the metric data store 72. Single observations (sometimes called "fine grained" data) can be blended into a single collective value. For example, in the exemplary illustrated embodiment, response times are averaged to determine a mean response time. Alternatively, a weighted mean or some other blending function could be used. The average response time is stored for later retrieval by the router computer 70 as described more fully below. When the average response time is retrieved, older observations are aged out of the metric data store. As a result, the average indicates server performance during a relatively recent time period, but variances in response time are dampened.

Since the metric tracker 126 was put in place without explicit instruction from the client program 84 or the server object 76, the metric data store 72 accumulates information about response time even though the client program 84 accesses the server object 76 without regard to whether the object 76 is load balanced. In addition, the server object 76 is freed from incorporating processing metric measurement logic. In this way, application software developers can more clearly focus on the business logic required for proper functionality of software applications incorporating server objects without considering process metric measurement logic.

Sharing Policies

In the above exemplary embodiments, a policy is used by a single object. However, objects can share policies. For example, an activator can be configured to place a second object in the same context as an existing object under certain circumstances (e.g., if two load balanced objects are of the same class). In other words, the two objects share the same object context property objects and associated policies, such as load balancing policies. In this way, a method call to either object results in automatic invocation of the appropriate shared policies to facilitate load balancing.

Load Balancing Service

Turning now to Figure 7, a load balancing service 502 on a router computer 70 provides load balancing services for client and server computers running in a networked environment. The service 502 includes one or more load balancing engines 62 and 64; the architecture supports a separate load balancing engine per load balanced object class. The load balancing engines 62 and 64 analyze processor metric data to update the global routing table 66. As described above, the global routing table 66 associates class identifiers with server computer identifiers, enabling the router activator 68 to determine which server computer should host an object creation request.

A catalog object 530 provides access to a central store of configuration information (not shown) regarding the server computers serviced by the router computer 70. The configuration information provided by the catalog object 530 includes values for the servers' network addresses, the servers' names (e.g., an identifier), server group names, and which servers are in what groups. The catalog object 530 additionally provides access to a list of applications designated as load balanced, a list of classes in each application, and a load balancing engine associated with each class, if any. A user interface is provided for editing the configuration information so that a system administrator can easily regroup the servers, add a server to the group, designate a load balancing engine for an object class, or perform other administrative tasks. The user interface is described in more detail below.

When the load balancing service 502 is initialized, a supervisor thread queries the catalog object 530 to determine the server groups serviced by the router computer 70. The supervisor thread creates server group objects 520 and server status objects 524 accordingly. The server status objects 524 store information relating to various server attributes, such as server name, server status (e.g., "not responding") and server network address. The server group objects 520 contain information about a particular set of servers serviced by the router computer 70. The group of servers is also sometimes called a target group. The load balancing service 502 can accommodate more than one target group simultaneously and keeps a server status object 524 for each server in a target group. Additionally, the router computer 70 can itself be a server computer in a target group.

During initialization, the supervisor thread creates instances of load balancing engines 62 and 64 as indicated by the catalog object 530 and builds a global routing table 66 of applications, classes, references to target groups (e.g., server group 520) and references to load balancing engines 62 and 64. Object class identifiers (CLSIDs) are associated in the global routing table 66 with a server computer identifier (or name), which identifies a particular server computer. Load balancing engines are passed a reference to the target groups for which they are responsible.

The load balancing engines 62 and 64 in the load balancing service 502 process and analyze the collective metric data store 504 to decide which server computers in a target group should host future object creation requests. The load balancing engines 62 and 64 then manipulate the global routing table 66 accordingly using a reporter object 522. Accordingly, a reference to the reporter object 522 is provided to the load balancing engines 62 and 64 upon initialization.

The router activator 68 runs in a service control manager 106 and shares the global routing table 66 with the reporter objects 522. When the router activator 68 receives an object creation request from a client computer 88, it forwards the request to an appropriate server in the global routing table as explained in the above discussion relating to creating an object.

A default load balancing engine 62 analyzes method call response time observations provided by the server computers and designates a server for hosting future object creation requests. Typically, the load balancing engine 62 modifies the global routing table 66 so that object creation requests of a particular object class are sent to a server in the target group reporting the lowest (i.e., fastest) response time for the object class. However, a different load balancing engine 64 could make decisions based on processing metrics other than response time and choose a preferred server computer accordingly. Additionally, the architecture accommodates a different load balancing engine per class of software objects. For example, the default method response time load balancing engine 62 could be designated for one class of objects, while a load balancing engine routing objects to a server having the most number of open database connections might be designated for another class of objects.

In an alternative embodiment of the invention, the load balancing engines 62 and 64 communicate with each other. Such an arrangement is useful, for example, to communicate information about object groups that tend to run in a particular sequence. For example, if it is known that an order submission object typically creates a credit check object, and the credit check object requires an open database connection to a credit bureau, the credit check object load balancing engine could communicate with the order submission object load balancing engine so that order submission objects are created on a machine that has an open database connection.

The metric collectors 506 are also created when the load balance service 502 is initialized. The metric collectors 506 are implemented in the illustrated embodiment as executing in a collection of threads, one thread for each server object (each server computer); however, if a target group is rather large, fewer threads (e.g., one) can be used. The metric collectors 506 have a list

of CLSIDs and create metric servers 508 on the server computers. The metric servers 508 in turn create a metric data store 72 on each of the server computers 78 and consolidate statistics about a particular processing metric for a given class on the server 78. The metric collectors 506 periodically poll the server computers' metric servers 508 to retrieve the consolidated metric data from the server computers' metric data stores 72, and the data is collected into the collective metric data store 504. The polling is achieved using a DCOM RPC remoting facility. The collective metric data store 504 is implemented as a shared memory location accessible by the metric collectors 506 and the load balancing engines 62 and 64.

A benefit to the polling arrangement is that between polls, plural observations can be blended to dampen variance in processing metric observations. In this way, the load balancing service 502 avoids making decisions based on spurious fluctuations typical in certain processing metrics. Accordingly, a system administrator can adjust the polling rate to tune overall system performance. Generally, a greater delay between polls results in greater variance dampening.

Alternative Routing Functions

Typically, the load balancing service routes requests based on the class identifier of the object creation request. However, the load balancing service can be configured to route requests based on some other function. For example, object creation requests can be routed based on the requesting client computer's identity, past routing of requests for a client based on the client computer's identity (also called client affinity), execution environment information associated with the requesting process (e.g., whether the process requires a single threading model or a transaction) or context data associated with the requesting client. In addition, the server can contribute to the routing determination by dynamically joining or leaving a target group. In other words, the server group members can change over time. For example, a server computer might instruct the router computer to remove it from a target group because the server computer has lost a database connection.

Implementation of Load Balancing in Alternative Layers

Although the above exemplary embodiments implement load balancing by using policies, the load balancing logic in the policies could be moved into a different layer. For example, the logic could be placed in the RPC layer or in a stub layer. Also, the layer could be intermittent. In other words, a statistical sampling method could be used to intercept and monitor a portion of method calls to collect processing metric data on a random or otherwise periodic basis instead of using a uniform monitoring method.

Just-in-Time Activation and Object Recycling

In the above examples, the server activator 134 (Fig. 3) fulfills an object creation request by arranging for creating a new instance of the server object 76. In some cases, however, the operating system may instead provide a reference without actually creating a new instance of the server object. Some calls (e.g., addref, release, queryinterface) can be handled by the operating

system and are processed without instantiating a new instance of the server object. Later, when the reference is required to complete a method call, an actual server object is created (or recycled, as explained below). Thus, the initial reference provided to the client program might not point to a server object. To prevent use of a reference pointing to no object, the operating system monitors the references and creates or recycles objects accordingly. These references are therefore sometimes called "safe" references, and the process of providing an actual object for a safe reference is sometimes called "activation." The term "just-in-time activation" describes delaying providing an actual object until necessary (e.g., when the client program requires a method call accessing the functionality of the object). The delay occurs between instantiation and the first method call.

Between method calls, an object typically retains state until it finishes its task (e.g., it completes or aborts a transaction); however, after the object finishes its task, it is deactivated (i.e., the object's state is destroyed) and sent to an object pool. Objects from the pool can be recycled for later use. In this way, system resources are conserved because recycling an object requires fewer resources than creating a new one.

Rebalancing Objects

Under certain circumstances, such as in the just-in-time activation scenario described above, a client may retain a reference to an object on a server that has since become overburdened. For example, a client may request an object and receive a safe reference to an object on a favorable server. At this point, an object has not actually been created, and the reference points to no object. Subsequently, when the client program tries to use the object, it may be that no objects can be recycled from the object pool, and the server has become severely overloaded. Thus, creating a new object on the overloaded server would result in unacceptable performance. Better performance would be realized if the object were instead created on a different server.

Separately, a client program may retain a reference to an object between transactions to avoid having to reestablish a connection. While such behavior may be appropriate under certain circumstances, performance on the server may degrade to unacceptable levels between the transactions. In such a case, better performance would be realized if the reference were released and an object created on a different server.

To address problems associated with maintaining references to overloaded servers, the architecture provides a rebalance message arrangement. With reference now to Figure 8, an object cluster on the server computer 78 includes a server object 76, a stub 74, an object context object 128, and an object policy set 140. The object cluster is assembled by an appropriate activator as described above. The client program runs in a client process 166 on the client computer 88 and accesses the server object 76 via a proxy 164.

The object policy set 140 of the server object 76 includes a retry policy 645. In the illustrated example, the server object 76 might not yet exist because the reference may be a safe

reference pointing to no object. The server-side retry policy 645 includes a method invoked before (or alternatively, after) method calls to the server object 76 and has access to the metric data store 72 and information from the other object policies 142-144 to determine if server performance is degraded or if certain server resources are overtaxed. On the client computer 88, an appropriate activator has included a retry policy 652 when assembling the object policy set referenced by the proxy 164.

A flowchart for a method of sending rebalance messages is shown in Figure 9. The rebalance messages are sent in the context of receiving a method call at the server stub 74, which transparently invokes the server-side retry policy 645 (Fig. 8). At step 702 (Fig. 9), the server-side retry policy 645 compares the processing metric value provided by the metric tracker 126 with an unacceptable level. The unacceptable level is a predetermined value adjustable by an operator at each server or controlled centrally at the router computer 70 (Fig. 3). If the level is determined not acceptable at step 704 (e.g., the response time is above a predetermined number of seconds), the server-side retry policy 645 sends a rebalance message to the client-side retry policy 652 at step 706. The client-side retry policy 652 then breaks the proxy's 164 connection to the stub 74 at an appropriate time at step 708. For example, the proxy 164 may have information indicating that the state of the server object 76 is not important (e.g., the object 76 is in its initial post initialization state) or the server object may be stateless. Accordingly, breaking the connection would not negatively impact the client program's use of the server object's functionality. Communication between the server-side retry policy 645 and the client-side retry policy 652 is achieved by passing a buffer using functionality provided by the DCOM RPC remoting facility. Alternatively, the stub 74 could produce a result indicating that the server computer 78 has failed, forcing the client program 84 to request another object.

At step 710, an object creation request is sent to the router computer as was described above, allowing reconnection to a server object on a different server computer. The object creation request can optionally be delayed until the client program 84 attempts to access the functionality of the server object, thus avoiding creating a new object until its functionality is required.

Neither the client program 84 nor the server object 76 needs to include logic to handle the rebalance message because the logic is accomplished by object policies 645 and 652 put in place by activators transparently to the client program 84 and the server object 76. The activators may be configured to assemble retry policies appropriate to the particular network environment, and their configuration may be changed without modification to the client program or the server object. Central management of the unacceptable performance level is achieved by configuring the router computer to set an appropriate activation property to a specified level value. The value is passed to the server activator, which provides it to the retry policy when assembling the server object policy set. An alternative arrangement would be to pass the value to the metric server 508 (Fig. 7), which could place the value in the metric data store 72 for access by the metric tracker 126 (Fig. 8). Still

another arrangement would be to store the value in a catalog available at each server computer and configurable by a system administrator from a central location.

Alternative Target Groups

In the above examples, the illustrated target groups are composed of server computers, and the router selects an appropriate server computer to host object creation requests. However, the load balancing architecture is extensible to support other target groups, such as threads, processes, and contexts.

For example, if an object requires any of a particular set of certain threads on which to run, the certain threads are placed in a target group associated with a class identifier. The load balancing service sends the object to a particular thread from the target group based on the object's class identifier.

Processing Metrics

In some of the above examples, a metric tracker observes response time and a load balancing engine analyzes response time to determine on which server objects should be created. Alternatively, various other processing metrics could be used to generate values generally indicative of performance. For example, the metric tracker might observe the number of recyclable objects in the server computer's object pool, whether a server computer has activated objects of a particular class identifier, the number of free database connections available to the server, load on a database at the server, or latency associated with the server's access to a database. In addition, metrics can be combined and weighted, resulting in a custom processing metric. For example, if two servers tie on response time, the tie could go to a server with the most recyclable objects in its pool. Another approach is to track only a particular method call to an object, or give various method calls a different weight. In this way, a method call most impacting perceived performance (e.g., a method for accepting a customer order) could be given more weight when making load balancing decisions.

In an alternative embodiment of the invention, the load balancing architecture is tailored to route objects known to produce a low variance in a processing metric to one group of servers; objects producing high variance are routed to another group. In this way, consistent performance of the overall system can be tuned more effectively. Since the high variance objects are isolated to a particular group, the objects' behavior can be more easily observed and additional steps taken to reduce the variance.

In addition, a processing metric might track the state of the object. For example, if several software objects on a server are performing a particular task (e.g., an order object waiting for a credit check), the load balancing engine may route object creation requests elsewhere. Since the metric tracker and load balancing engine can be tailored to the specific behavior of the object, the two can cooperate to balance the processing load among servers more effectively. Since the load balancing logic is separated from the server objects implementing the business logic of the

application, various engines and trackers can generally be prototyped or tested without introducing logic errors (i.e., software bugs) into the application. Finally, the load balancing engines in the illustrated exemplary embodiment are easily replaceable (sometimes called "pluggable"). Replaceability is achieved because the load balancing engines use a common defined interface to access processing metric data and modify the global routing table.

User Interface

With reference now to Figure 10, the window 800 shows a user interface by which an administrator can set load balancing for an object. The interface presents a list of objects from which the administrator can select. For example, an administrator might select the object BANK.ACCOUNT 808. A second window 810 is then presented, from which the administrator can enable load balancing for the object, choose a load balancing engine and specify a target group. In addition to providing the illustrated per object load balancing, the user interface also provides a way to enable load balancing for an entire application. The catalog is updated accordingly to reflect the administrator's load balancing selections. In addition, the catalog can be pre-configured by an application developer to relieve the administrator of certain configuration duties. Various related settings such as the appropriate load balancing activator, a polling interval and a retry interval are preferably stored in the registry.

Contexts, Policies, and Activators

The following sections describe an exemplary implementation of contexts, policies, and activators for use with load balancing.

Component Application Execution Environment

With reference to Figure 11, component application objects 1110-1118 execute on computers 1102-1103 (such as the computer 20 of Figure 1) within an extensible object execution environment 1100 of the COM+ services that can be extended to incorporate additional domain-specific behaviors according to an illustrated embodiment of the invention. In the illustrated environment 1100, the component application objects 1110-1118 each execute within particular domains. The domains are independent aspects of the environment that contain a group of like-aspected component application objects (e.g., environment aspects that cause the objects to be treated alike by an environment behavior specific to the domain), and effectively establish a boundary about the group.

Domains in the illustrated environment 1100 include a locality domain, which is a machine (e.g., computers 1102-1103), process (e.g., processes 1126-1128) and apartment (e.g., apartments 1120-1123) in which the object resides. Accordingly, each apartment 1120-1123 of the illustrated environment 1100 is a separate domain. As is well known in the industry, a process is an executing program state in the Microsoft Windows NT and like operating systems that consists of a private virtual address space, program code and data, and other operating system resources,

such as files, pipes, and synchronization objects that are visible to the process. A process also contains one or more threads that run in the process. A thread is a basic entity to which the Microsoft Windows NT and like operating systems allocate processing time on the computer's processing unit (e.g., processing unit 21 of Figure 1) for instruction execution. An apartment is an entity in accordance with a threading model, such as the single-threaded apartment (STA) or multi-threaded apartment (MTA) threading models, among others. For example, an apartment according to the STA threading model consists of program code (e.g., a group of objects) that resides or runs on only a particular thread (e.g., the thread on which the object is created).

The domains in the illustrated environment 1100 also include other aspects of the execution environment of the objects 1110-1118, such as security role for a role-based security behavior or transaction for an automatic transaction behavior of the types provided in the MTS system. Exemplary such domains 1130-1132 are shown in Figure 11 as dotted lines in oval shape. The domains (including the locality domains that are coextensive with apartments 1120-1123 and the exemplary domains 1130-1132) may arbitrarily intersect and nest.

Contexts

With reference still to Figure 11, a context in the illustrated object execution environment 1100 conceptually is an intersection of the domains. For example, contexts 1140-1146 are formed at the intersections of the locality domains of apartments 1120-1123 with the exemplary other domains 1130-1132. Further, a set of one or more of the objects 1110-1118 in the same domains are in a same context. For example, the objects 1110-1111 are at the intersection of domains 1120, 1130 and 1131, and therefore are in the same context 1140. All of the component application objects 1110-1118 are in a context 1140-1146. Further, since the apartments 1120-1123 constitute domains of the illustrated environment 1100, the contexts 1140-1146 are all within (i.e., smaller or equal in scope to) the apartments 1120-1123.

With reference now to Figure 12, the contexts 1140-1146 (Figure 11) each are represented in COM+ as an object context object 1160. The object context object 1160 is implemented as a COM object, which is automatically provided by the COM+ system and associated with the component application objects in the context as described below. All objects 1110-1118 in the illustrated environment 1100 (Figure 11) are within a context, and have an association to the object context object 1160 that represents the context. This association is maintained at run-time by the COM+ run-time services, and the object context object associated with a component application object can be retrieved via a call to a "CoGetObjectContext()" API of COM+. Objects that are in a same context of the environment (e.g., objects 1110-1111 in context 1140, or objects 1117-1118 in context 1145) share a same object context object 1160 that represents the context.

The object context object 1160 encapsulates a set of context properties that characterize the domains whose intersection forms the context. More specifically, the object

context object has a property table 1162, which is an ordered list of (context property identifier, context property object reference) value pairs. The context property identifier is a globally unique identifier (GUID) that identifies the context property. The context property object reference is a pointer to a context property object 1164-1167 that represents the domain's characteristics. The contexts 1140-1146 of the component application objects 1110-1118 in the illustrated environment 1100 are immutable during the component application objects' lifetimes. Accordingly, once the object context object for a context is set up (generally at instantiation of a first object in the context), the context properties of the object are frozen to prevent any changes. In alternative implementations of object context objects, the context properties need not be frozen.

The object context object 1160 supports an "IObjectContext" interface 1168 as defined in the program listing 1170 shown in Figure 13. The "IObjectContext" interface 1168 provides member functions to set or get the context property object for a given context property identifier, and to enumerate the context property objects of the object context object in order. The "IObjectContext" interface 1168 also provides a member function to prevent further modification of its context properties (i.e., to freeze the context) after set-up.

Policies

With reference to Figures 14 and 15, the environment behaviors specific to the domains (including the apartments 1120-1123 and other domains 1130-1132) in the illustrated environment 1100 (Figure 11) are realized in part by policies that respond to context events triggered on calls made between objects in separate contexts (i.e., across context boundaries). The context events allow the policies to respond automatically when the logical execution sequence of the component application crosses a context boundary to enforce the domain-specific behaviors, without the component application objects having to explicitly invoke the behaviors (e.g., in the programming of the component application's code). The policies thus implement the semantics of entering or leaving a domain. In response to a context event, the policy can perform various processing of its domain-specific behavior, such as having a side effect (e.g., initiate a transaction, such as in the automatic transaction behavior), passing information to its partner policy in the other context, failing the call, forcing a different return value, or short-circuit the call (e.g., acting as a handler).

The policies are implemented as policy objects 1180, which are COM objects denoted by triangles in Figures 14 and 15. The policy objects 1180 all support an "IPolicy" interface (defined as shown by the program listing 1181 of Figure 16), which acts as a sink for the context events. The context events include context events that are delivered to policies on a client-side of the call (i.e., the object that issued the call), which include "call" and "return" context events (herein termed "client-side context events"). The context events delivered to policies on a server-side of the call include "enter" and "leave" context events (herein termed "server-side context events"). The policy objects can send a buffer of data to a counterpart policy object of a same

domain on the other side of the call, which is delivered to the counterpart policy object with the context events for its respective side of the call. Context events also are delivered to policy objects at times other than during a call to the server object. Specifically, the program listing 1181 defines a "ReleasePolicy" context event which is delivered to server object's policy objects when a policy object is removed from a policy set (described below), and an "AddRefPolicy" context event which is delivered when a policy object is added to a policy set.

Cross-Context Object References And Policy Sets

In COM+, a reference to an object in another context (herein termed a "cross-context object reference" 1182-1183) is indirect via an object context switcher. For example, a reference held by the object 1110 in the context 1140 (Figure 11) to the object 1112 in the context 1141 (Figure 11) is a cross-context object reference via an object context switcher. A reference to an object in a same context is a direct pointer. For example, the object 1117 in the context 1145 (Figure 11) can hold a reference to the object 1118 also in the context 1145 (Figure 11) which is a direct interface pointer.

In the case where a client component application object 1190 is in the same apartment as a server component application object 1192 (i.e., a same apartment, cross-context reference 1182), the object context switcher is a wrapper 1184 (Figure 14). The wrapper 1184 is a light-weight type proxy in the sense that the wrapper 1184 does not perform marshaling (i.e., transferring data, such as call parameters, between address spaces of the client and server localities, such as the virtual address spaces of different processes or computers). Instead, the wrapper 1184 simply performs processing for the context switch, such as by issuing context events to the policies 1180. In the case where the client and server component application objects 1190, 1192 are in different apartments or processes (i.e., a cross-apartment or process, cross-context reference 1183), the object context switcher is implemented as a proxy 1186 and stub 1187 pair, which performs marshaling as well as context switching. For marshaling, the proxy 1186 and stub 1187 pair utilize the COM RPC Standard Marshaling Architecture of OLE and DCOM, which is described, *inter alia*, in Brockschmidt, *Inside OLE, Second Edition* 277-340 (1995), with added support for contexts as described below.

Further, the cross-context object references 1182-1183 have associated policy sets 1194-1196. The policy sets 1194-1196 are collections of the policy objects 1180 that receive context events for calls from a specific client context 1198 to a specific server context 1199 (e.g., from the context 1140 to the context 1141 in Figure 11). The COM+ services maintain a single policy set to receive the context events for calls from a given originating context to a given destination context, which is shared by all cross-context references held by objects in the originating context to objects in the destination context. A different policy set may be maintained for references in the reverse direction. In the case of the same apartment, cross-context reference 1182, the wrapper 1184 has a single policy set that contains both client-side and server-side policy

objects 1188-1189. In the case of the cross-apartment or process, cross-context reference, the proxy 1186 and the stub 1187 have separate policy sets for the client-side policy objects 1188 and the server-side policy objects 1189, respectively.

When a call is made on the cross-context object reference, the object context switcher delivers context events to the policy objects 1180 in the policy set 1194-1196, both before (for the call and enter events) and after (for the leave and return events) the object context switcher passes the call through to the server object. The policy objects 1180 in the policy set are composed in some order. The object context switcher delivers the context events to the policy objects in this order on the call and enter events, and in reverse order for leave and return events.

With reference to Figure 17, the policy sets 1194-1196 (Figures 14 and 15) each are implemented as a policy set object 1200 (a COM object) that encapsulates an ordered list 1202 of policy records. Each policy record contains an event mask, policy identifier, and policy object reference for a particular policy object 1180 in the policy set. The event mask is a set of Boolean flags that indicate which context events are to be issued to the respective policy object during a call on a cross-context reference having the policy set. The policy identifier is a unique identifier (a GUID) that indicates the policy represented by the policy object 1180. The policy object reference is an interface pointer to the policy object 1180. In some embodiments of the invention, the policy set object 1200 can be implemented in a same object as the object context switcher (e.g., the wrapper 1184 or proxy 1186 and stub 1187). Alternatively, the policy set object 1200 can be a separate object with a private interface to the object context switcher, through which the policy set object 1200 is notified of the context events to issue to its policy objects 1180.

The policy set object 1200 supports an IPolicySet interface 1206 shown in the program listing 1208 of Fig 18. The IPolicySet interface 1206 has member functions to build the collection of policy objects 1180. An "AddPolicy()" member function specifies information for a policy record to add to the policy set object's ordered list 1202. In some alternative implementations, the "AddPolicy()" member function can contain ordering information that specifies where the added policy is to be placed in the order of the policies in the policy set. When called, the "AddPolicy()" member function causes the "AddRefPolicy" context event to be delivered to policy objects in the policy set. A "Freeze()" member function is called when building the policy set is completed, which prevents any further additions to the ordered list 1202.

Policy Makers

With reference to Figure 19, the object context object 1160 (Figure 12) contains policy makers that dynamically construct the policies 1180 and policy sets 1194-1196 for the cross-context references 1182-1183 (Figures 14 and 15). In COM+, the policy makers are simply those of the context property objects 1164-1167 that implement an "IPolicyMaker" interface defined in a program listing 1210 shown in Figure 19. At reference transfer time (more specifically, on unmarshaling a reference to the server object into another context), the policy makers in the object

context objects of the client context 1198 and server context 1199 contribute policies 1180 (Figures 14 and 15) and determine their order in the policy set 1194-1196, as described more fully below. As noted above, the context property objects 1164-1167 listed in an object context object 1160 (and therefore also its policy makers) is static and immutable after creation and set-up of the object context object.

The IPolicyMaker interface includes an "AddPolicies()" member function that is called when a cross-context reference from or to the context represented by the object context object 1160 is formed. The "AddPolicies()" member function determines whether to add any policy objects for its respective context property to the policy set for the cross-context reference, and the context events that such policy object(s) should receive. The parameters of the "AddPolicies()" member function specify the policy set to which any policy objects are to be added (the "pset" parameter), a value indicating the type of context marshaling (the "cm" parameter), and the other context of the cross-context reference (the "pctxtDest" parameter). The policy maker calls the "AddPolicy()" member function on the "IPolicySet" interface 1206 (Figure 18) of the policy set object 1200 (Figure 17) for each such policy object to add that policy object to the policy set. The policy makers in the object context object of the client side context 1198 (Figures 14 and 15) add only the client-side policy objects 1188 to the policy set 1194-1196, while policy makers for the server side context 1199 add only server-side policy objects.

Envoys

In the illustrated extensible object execution environment 1100, the policy makers in the server-side context 1199 also can contribute special policies called "envoy policy objects" to the client-side policy objects 1188 in the client-side policy set 1195 (Figure 15). The server-side policy maker can contribute such envoy policy objects to act as sinks for client-side context events. On receiving these events, the envoys can pass information from the client-side context to server-side policy objects 1189. In the illustrated environment, the server-side policy maker (or simply a context property object) that contributes an envoy policy object implements an "IMarshalEnvoy" interface. Such policy makers are called "envoys." In alternative implementations, the server-side policy maker may identify itself as an envoy to be marshaled into a client's context by setting a context marshaling flag (e.g., as shown in Figure 19) that indicates the policy maker is an envoy. During marshaling of a server object reference across apartments, processes, or machines, the server context object along with all its context property objects that implement the "IMarshalEnvoy" interface are marshaled along with the reference as part of the reference marshaling process described more fully below. When unmarshaled at the client-side, the envoys may contribute a client-side policy object to the client-side policy set. The envoy preferably is marshaled using copy-by-value marshaling, which does not add to the server-side context property object's reference count.

Cross-Context Reference Tracking Structures

With reference again to Figures 11 and 20, the COM+ runtime services 1220 (Figure 20) (provided by the COM+ component of the Windows NT 5.0 operating system as stated above) in the illustrated extensible object execution environment 1100 (Figure 11) maintain data structures to track object contexts for use during cross-context reference creation. In particular, the COM+ runtime services 1220 maintain a wrapper table, a policy set table and a registration information table per each process (e.g., the process 1126) in the environment 1100 (called the process-wide tables). These tables contain information on a process-wise basis to facilitate sharing common interface proxies for the server object across all client contexts.

Alternative implementations of the invention can use other data structures for object context tracking, such as multiple level indexes, caches, etc. In particular, some alternative implementations of the invention can maintain wrapper and policy set tables on a per context or other basis, rather than maintain tables used globally in each process. This is particularly useful in an alternative implementation of the invention in which a separate wrapper to the server object is provided for each client context. In the case of per context tables, a global table or set of tables indexed by client context may provide an initial look-up of the appropriate per context wrapper and policy set tables.

Wrapper Table. As described more fully below and shown in Figure 21, the COM+ runtime services 1220 use a single wrapper object 1230 to implement the same-apartment, cross-context references 1184 across all client contexts inside the server object's apartment. The wrapper table 1222 tracks on a process-wide basis an association (or mapping) of the server objects to their respective wrapper objects. The server object's identity is represented by the server object's "IUnknown" interface 1231 pointer. This pointer is used as an index into the wrapper table to return the server object's associated wrapper object 1230.

Policy Set Table. Given a client context and server context pair in the illustrated environment 1100, there exists a unique policy set that contains client-side and server-side policy objects to which client-side and server-side call events are respectively delivered when a call is made from the client context to the sever context. The policy set table 1224 provides a mapping of a pair of contexts in the illustrated environment 1100 to a policy set that provides the context switching between the contexts. The policy set table 1224 thus is indexed by a client and server context pair ("CCTXT" and "SCTXT") to look up their respective policy set. The context pairs to policy set relationships maintained by policy set table are not symmetric. More particularly, two contexts A and B may map to a different policy set when A is the client context and B is the server context, than when A is the server context and B is the client context.

Registration Table. The registration table 1226 tracks registration information as to the cross-context references held by clients to the server object. The registration table 1226 associates a client and server context pair together with the server object (identified by its "IUnknown" interface pointer) to this registration information. The registration information

contains the number of references and the list of interfaces held legally by clients in one client context on the server object in the server object's context. In some alternative implementations of the invention, the registration table can be integrated with the policy set table, so that the registration information for a client and server context pair is kept in one table together with the policy set information.

Cross-Context Reference Creation at Marshal Time

In the illustrated object execution environment 1100, the cross-context references 1182-1183 (Figures 14 and 15) are created by the COM+ runtime services 1220 (Figure 20) upon marshaling and unmarshaling an interface pointer to the server object 1192 into a client context (e.g., upon passing the interface pointer over to the client as an [in] or [out] parameter of a method call). These cross-context references 1182-1183 are implemented, in part, using the Standard Marshaling Architecture of the Microsoft COM Remote Procedure Call (RPC). (For a more detailed discussion of the Microsoft COM RPC, see Brockschmidt, Inside OLE, Second Edition 277-338 (Microsoft Press 1995)). (For expository convenience, the following discussion assumes the object is standard marshaled. However, the process described also applies to server objects that are custom marshaled.) This Standard Marshaling Architecture is modified in the illustrated environment to also create the cross-context reference data structures according to the invention as described below. The following discussion explains in greater detail the various steps involved in marshaling and unmarshaling for each combination of destination context (e.g., same apartment, cross apartment, and cross process) and interface pointer type (e.g., wrapper, proxy and naked). Figures 21-23 show the relative placement of context-specific data structures with respect to the COM RPC Standard Marshaling Architecture.

For purposes of the following cross-context reference creation discussion, an interface pointer is termed a "proxy" when it is representing an interface on a server object that resides in a different apartment. In the illustrated execution environment 1100, an interface pointer that is a proxy supports an "IStdManager" interface 1258. An interface pointer is termed a "wrapper" when it represents an interface on a server object that resides in a different context but within the same apartment. An interface pointer that is a wrapper supports an "IWrapper" interface 1244. An interface pointer that is neither a proxy nor a wrapper is termed "naked".

Same Apartment, Cross-Context Case. With reference to Figure 21, the wrapper 1184 of the same apartment, cross-context reference 1182 (Figure 14) is created by the COM+ runtime services 1220 (Figure 20) when a native interface pointer to the server object 1192 is marshaled from the server object's context 1199. The wrapper 1184 includes a wrapper object 1230, a set of interface proxies 1232-1233 (called "facelets") and interface stubs 1236-1237 that correspond to individual interfaces 1234-1235 on the server object 1192, and context channels 1238-1239. The wrapper object 1230 acts as a proxy manager in the COM RPC Standard Marshaling Architecture to manage the facelets 1232-1233 and stublets 1236-1237. In accordance

with the Standard Marshaling Architecture, the facelets and stublets are generated by a MIDL compiler from interface definition language descriptions of the server object's interfaces. The facelets 1232-1233 are generated to provide interfaces 1242-1243 that match respective interfaces of the server object 1192 (e.g., with an identical virtual function table structure 202 of Figure 4), however the facelets' underlying implementation of the interfaces' methods operate to pass the client's method calls through the context channel to the stublets. The stublets 1236-1237 then issue the method calls to the actual interfaces 1234-1235 on the server object 1192. In the same apartment case, the facelets and stublets are termed "light-weight," meaning they do not perform marshaling and unmarshaling of call parameters and associated data into an RPC. The context channels 1238-1239 implement context switching between the client's context 1198 and that 1199 of the server object 1192 by dispatching policy events to the appropriate policy set for the client and server context pair. The context channels 1238-1239 and the stublets 1236-1237 support an "IRpcChannelBuffer" interface 1240 and an "IRpcStubBuffer" interface 1241 (which are standard interfaces defined per the COM RPC Standard Marshaling Architecture), respectively, which are used to pass the client's method invocations through to the server object 1192.

On marshaling the native interface pointer to the server object 1192 from the server object's context 1199, the COM+ runtime services utilize the server object's "IUnknown" interface pointer to look up an existing wrapper object 1230 for the server object 1192 from the wrapper table 1222 (Figure 20). The COM+ runtime services uses the same wrapper for all contexts in the same apartment as the server object, and thus uses the wrapper object 1230 identified in the wrapper table 1222 if any exists. Otherwise, when an existing wrapper is not found, the COM+ runtime services create a new wrapper object 1230 for the server object and adds a wrapper table entry that maps the server object's IUnknown interface pointer to the new wrapper object 1230 into the wrapper table 1222. (Some alternative implementations of the invention may provide a different wrapper to the server object for each client context. In which case, the runtime services look up the wrapper to the server object for the particular client context. If not found, the wrapper is created and an entry added to the client context's wrapper table.)

After the wrapper object 1230 is identified or created anew, the COM+ runtime services 1220 checks whether the facelet, context channel and stublet data structures corresponding to the interface pointer being marshaled have already been created. If not, the COM+ runtime services 1220 creates these data structures. The COM+ runtime services 1220 then completes the marshaling of the native interface pointer from the server object's context 1199 by substituting a direct pointer to the corresponding facelet's interface 1242-1243.

When the interface pointer is then unmarshaled in the client context 1198, the COM+ runtime services issue a query interface call using the unmarshaled interface pointer to request an "IWrapper" interface. Since the facelet 1232-1233 is aggregated into the wrapper object 1230, this call returns an interface pointer to the "IWrapper" interface 1244 on the wrapper object 1230. The

"IWrapper" interface is defined as shown in the program listing 1245 in Figure 24, and includes a "LookupChannel" method that provides a lookup of the context channel 1238-1239 of the facelet 1232-1233. Using the "LookupChannel" method on the "IWrapper" interface, COM+ obtains a pointer to an "ICtxChannel" interface 1248 on the context channel 1238-1239 behind the facelet 1232-1233. The "ICtxChannel" interface is defined as shown in the program listing 1249 in Figure 25.

With the "ICtxChannel" interface pointer, the COM+ runtime services then call a "Register()" method on the context channel 1238-1239 to inform the context channel that the interface represented by the facelet to which it is connected is being unmarshaled in the client context 1198. In the "Register()" method, the context channel 1238-1239 obtains the client context 1198 using a "CoGetCurrentContext" API. The context channel then looks up the policy set of the client and server context pair from the policy set table 1224 (Figure 20), to which the context channel delivers policy events during method invocations on the reference 1184. If no policy set associated with the client and server context pair exists, the context channel creates the policy set using the contexts' policy makers as described above and adds the policy set into the policy set table. The context channel also adds information into the registration table indicating that the facelet interface 1242, 1243 has been unmarshaled into the client context 1198.

Because the facelets 1232-1233 for all same apartment, cross-context references to the server object 1192 are aggregated into the wrapper object 1230, any "AddRef()" and "Release()" method invocations on the facelet interfaces 1242-1243 are passed to the wrapper object 1230. The wrapper object's implementations of these methods update the registration information in the registration table 1226 to reflect the number of references to the server object held by clients in the client context 1198.

On marshaling of the wrapper object or facelets into another client context in the apartment (e.g., when the client passes its reference to the facelet interfaces 1242-1243 as a parameter in a call), the same-apartment, cross-context reference to the server object is set up similarly to the marshaling of a direct interface pointer to the server object 1192 just described. In the case of the wrapper object or facelets reference, however, no marshaling is involved.

Cross-Apartment Case. With reference to Figures 22 and 23, the proxy 1186 and stub 1187 of the cross-apartment, cross-context reference 1183 (Figure 15) is created by the COM+ runtime services 1220 (Figure 20) when a native interface pointer to the server object 1192 is marshaled from the server object's context 1199 and unmarshaled into a client context 1198 in another apartment (whether in the same process, another process or another machine). The proxy 1186 and stub 1187 use standard marshaling per the COM RPC Standard Marshaling Architecture to remote method invocations between the apartments, with the addition of data structures to provide context switching during the method invocations.

More specifically, the data structures of the proxy 1186 (Figure 22) for the server object reference 1183 includes a proxy manager 1250 to manage the facelets 1232-1233 for the server object 1192 as well as DCOM channels 1252-1253 as per the conventional COM RPC Standard Marshaling Architecture. The stub 1187 (Figure 23) includes a stub manager 1260, a DCOM channel 1262, and the stublets 1236-1237 of the server object's interfaces 1234-1235 also per the conventional COM RPC Standard Marshaling Architecture. The illustrated proxy 1186 and stub 1187 add a context channels 1254-1255 and 1264 to the conventional Standard Marshaling Architecture before the DCOM channels 1252-1253 on the client-side and behind the DCOM channel 1262 on the server-side. Like the context channels 1238-1239 (Figure 21) for the same-apartment case, the context channels 1254-1255 and 1264 deliver policy events during the client's method invocations to the server object. Only, the context channels 1254-1255 in the proxy 1186 deliver client-side policy events to the client-side policy set 1195 (Figure 15), while the context channel 1264 in the stub 1187 deliver server-side events to the server-side policy set 1196 (Figure 15). The proxy's context channels 1254-1255 also provide processing of cross-context marshaling of the proxy 1186 within the client's apartment. In other words, the same remote proxy 1186 is used for all contexts in the client apartment.

The conventional data structures of the proxy 1186 and stub 1187 are created using the usual COM RPC Standard Marshaling Architecture process. The COM+ runtimes services add extra steps to this process to create the context channels 1254-1255 and 1264, and the client- and server-side policy sets 1195-1196, as well as registering in the COM+ runtime services' tables (Figure 20). Specifically, upon marshaling a naked interface pointer from the server context 1199 to another apartment, the COM+ runtime services create the context channel 1264 (Figure 23) and the server-side policy set 1196 (Figure 15). The COM+ runtime services also register the server-side policy set 1196 in the policy set table 1224. Further, a pointer to the server context object (which is required to be apartment agile, meaning the object can be called directly from any context or apartment in the process) is marshaled with the server object interface pointer to the client apartment for use in generating the client-side policy set. Context property objects of the server context that are envoys (described above) also are marshaled to the client context 1198 for adding to the client-side policy set 1194 (Figure 15).

On unmarshaling the interface pointer in the client apartment, the COM+ runtime services create the context channels 1254-1255 (Figure 22) and the client-side policy set 1195 (Figure 15), including adding envoys marshaled from the server context 1199. The COM+ runtime services also register the server-side policy set 1195 in the policy set table 1224 (Figure 20).

Wrapper and proxy type interface pointers on the server object 1192 are marshaled and unmarshaled cross-apartments similarly to the cross-apartment marshaling of a naked interface pointer just described. For a wrapper interface pointer, the wrapper object 1230 obtains a naked

interface pointer on the server object 1192, and delegates to marshaling the naked interface pointer via the Standard Marshaling Architecture as just described. For a proxy interface pointer, the proxy interface pointer is marshaled in such a way as to produce a same marshaled representation (called an "OBJREF") of the interface pointer as would be generated when marshaling a naked interface pointer to the server object. The proxy interface pointer can then be unmarshaled in the same manner as the naked interface pointer.

Cross-Process And Cross-Machine Cases. The cross-process and cross-machine marshaling and unmarshaling of server object interface pointers is similar to that of the cross-apartment case except that the "OBJREF" marshaled representation of the interface pointer is generated to also contain the marshaled server context for use in forming the client-side policy set.

The following Table 1 summarizes the processing to marshal and unmarshal the various interface pointer types for each of the above described destination context types.

Table 1. Marshaling And Unmarshaling Of Server Object Interface Pointers.

	NAKED	WRAPPER	PROXY
Cross-Context	Marshaling: • Create/find Wrapper Unmarshaling: • Obtain Context channel • Create/find policy set • Add registration info	Marshaling: • None Unmarshaling: • Obtain Context channel • Create/find policy set • Add registration info	Marshaling: • None Unmarshaling: • Obtain Context channel • Create/find policy set • Add registration info
Cross-Apartment	Marshaling: • Create/find stub manager • Create/find remote policy set Unmarshaling: • Create/find proxy manager • Create/find policy set • Add registration info	• Obtain naked pointer • Follow steps for naked pointer	Marshaling: • Marshal proxy manager Unmarshaling: • Create/find proxy manager • Create/find policy set • Add registration info

Table 1. Marshaling And Unmarshaling Of Server Object Interface Pointers.

	NAKED	WRAPPER	PROXY
Cross-Process / Cross-Machine	Marshaling: • Create/find stub manager • Create/find remote policy set • Assign ContextId to server context • Marshal server context Unmarshaling • Create/find proxy manager • Unmarshal server context • Create/find policy set • Add registration info	• Obtain naked pointer • Follow steps for naked pointer	Marshaling: • Marshal proxy manager • Marshal server context Unmarshaling • Create/find proxy manager • Unmarshal server context • Create/find policy set • Add registration info

Method Invocations. The following discussion briefly explains the sequence of events that occur during method invocation by the client using the server object references 1182-1183 (Figures 14 and 15). These calls can be broadly classified into two groups, cross-context calls (via the wrapper 1184 in Figure 21) and remote calls (via the proxy 1186 and stub 1187 in Figures 22 and 23).

Same Apartment, Cross-Context Calls. When the client 1190 makes a method invocation on the facelet 1232-1233 in the wrapper 1184 (Figure 21), the facelet generates a marshal packet and delivers the packet to the context channel 1238-1239 to which the facelet connects for onward transmission to the stublet 1236-1237. The context channel 1238-1239 obtains the client context 1198 using the "CoGetCurrentContext()" API and performing a lookup in the policy set table 1224 (Figure 20) to see if the interface reference has been legally passed to the client context. If this check fails, the context channel fails the call and returns a "CO_E_WRONG_CONTEXT" result. Otherwise, the context channel delivers call events in order to client-side policy objects in the client-side policy set 1188 (Figure 14), switches to server context 1199, delivers call events to server-side policy objects in the server-side policy set 1189, and invokes the method on the server object 1192 through the stublet 1236-1237. When the method invocation on the server object 1192 returns, the stublet generates a marshal packet and returns the packet to the context channel 1238-1239 for transmission back to the facelet 1232-1233. The context channel 1238-1239 delivers return events to server-side policy objects 1189, switches back

to the client context 1198, delivers return events to client-side policy objects 1188, and returns to the facelet. The facelet returns to the client after unmarshaling the packet.

In some alternative implementations of the invention, context event delivery can be configured so that context events are delivered to certain policy objects at different stages of the call. In one such implementation for example, context events can be delivered at each side of the call to particular policy objects in the stages: before marshaling, after marshaling, before synchronization, and after synchronization.

On an error during context event delivery, the context channel preferably causes all policy objects that have processed context events for the call to undo or reverse their context event processing. The context channel calls the policy objects that were delivered a context event during the call in reverse of the order in which the context events were delivered. A policy object can issue an error during the context policy delivery to initiate this error handling by calling a "nullify()" member function of the "IRpcCall" interface, a pointer to which is passed to the policy object with the context event as shown in Figure 16.

Remote Calls. The remote calls via the proxy 1186 and stub 1187 (Figures 22 and 23) are very similar to the cross-context calls except that the proxy's context channel 1254-1255 (Figure 22) delivers call events to the client-side policy objects 1188, leaves the client context 1198, and delegates the marshaled packet to the proxy's DCOM channel 1252-1253 for onward transmission to the server object's process. The DCOM channel 1252-1253 switches from the client apartment if necessary and delegates to the RPC service of the COM RPC Standard Marshaling Architecture. When the marshaled packet arrives at the server's process, the packet is delivered to the stub's DCOM channel 1262, which switches to the server object's apartment. The stub's DCOM channel 262 then delivers the marshal packet to the context channel 1264 through the "ICtxChannel::Invoke()" method. The context channel 1264 switches to the server object's context 1199, delivers call events to the server-side policy objects 1189 (Figure 15) and invokes the method on the server object 1192 through the stublet 1236-1237. The sequence of events is reversed on return from the method.

Activators

With reference to Figure 26, the object context object 1160 (Figure 12) of a context in the illustrated execution environment 1100 (Figure 11) is created at instantiation of a first object in the context by activators 1352-1357. In COM+, the activators 1352-1357 provide extensible object instantiation services via delegation of an object instantiation request through a potentially distributed chain 1350 of activators (termed an "activator chain" 1350). The activator chain 1350 is responsible for activating a component application object (e.g., a server object 1192) of the requested class in the proper location and context (e.g., server context 1199). Individual activators in the chain 1350 determine a context for a new object, select or create a location and context for the new object, as well as create and return an interface pointer to the new object in that context.

In COM+, the activators 1352-1357 are implemented as COM objects that support an "ISystemActivator" interface (defined in the program listing 1370 shown in Figure 28, and described below).

Activation begins at an object instantiation service (e.g., the "CoCreateInstance()" API) that receives an object instantiation request for a component application object of a specified class from a client (e.g., a client component application object 1190 or other client program). The object instantiation service processes standard COM activation logic, and can also delegate further activation processing over to another activator. The other activator can modify context properties of the context in which the component application object is to be instantiated, and further delegate to yet other activators (and so on, through plural activators forming the activator chain 1350). The activators to which the instantiation request is delegated can include custom activators in addition to COM+-provided activators. The activators may be specific to the requested component application object class, the client context (e.g., client context activators 1352 and server context activators 1357), the location (e.g., process activators 1353, 1356 and machine activators 1354, 1355), and other context properties.

The delegation through the activator chain 1350 decides the properties for the context in which the component application object of the requested class is to be activated, such as the machine, process, apartment, and other domains. In some cases, traversal of the activator chain 1350 results in the server component application object 1192 being created in the same context as the client object 1190 that requested its creation (i.e., the client context 1198). In other cases, the traversal results in the component application object 1192 being created in another context (i.e., a separate server context 1199). If the resulting context does not yet exist, the activator chain 1350 creates the server context 1199 (i.e., the object context object 1160 with appropriate context property objects 1164). In still other cases, the activator chain can terminate activation of the object, or defer activation. For example, the activator chain can provide the above-mentioned JIT activation behavior by creating a wrapper or stub of the object without actually instantiating the object itself, and defer completing activation until the wrapper or stub instantiates the object on receiving a call to the object.

In COM+, the activator chain begins with an initial delegation to an activator (termed the "immediate activator") by the "CoCreateInstance()" or other object creation API. This initial delegation by the "CoCreateInstance()" API is to a class-specific activator (i.e., specific to the server object's class) if any is designated for the class. The class-specific activator can implement specific processing for the class during instantiation, such as setting particular context properties when objects of the class are instantiated. Otherwise, if no class-specific activator is designated for the class, the "CoCreateInstance()" API delegates to a default activator of the client context 1198. The default client context activator can implement activation processing specific to a particular type of context. For example, contexts that incorporate particular domain-specific behaviors (e.g., a

context with the above-mentioned MTS system behaviors, or a context with particular threading model behaviors) can provide a default activator to set context properties specific to the behaviors (such as, to provide a "transaction" context property in MTS contexts). Finally, if the client context 1198 does not provide a default activator, the "CoCreateInstance()" API initially delegates to a default context activator.

Activation Stages. After the initial delegation to the immediate activator, activation delegation proceeds through a multiple stage sequence. The COM infrastructure provides access to Custom Activators at the following activation stages: Services Control Manager (SCM), process, apartment, and context. The stages are further qualified to Client, Router, Server. For example, Client SCM, Router SCM, Server SCM defines explicit SCM stages an activation may go through. The stages model the COM hierarchy for locating where an object is created. At each stage, custom activators that are specific to the server object class, in addition to COM+-provided activators, may be invoked. The custom activators influence the location (i.e., machine, process, apartment, and context) where the server object is created, as well as initializing properties of the server object's context. In alternative implementations of the invention, the client also can contribute custom activators to which activation is delegated in particular of the stages.

The process of activation goes through the various stages in a particular order, called a "chain" or "journey." An earlier stage delegates to the current stage that may delegate on to the next stage, as shown in the following Table 2. What activator is invoked when the current activator delegates onward is controlled by the COM+ infrastructure. The notation used in the following Table 1 is: <stage>A(<side>) where the <stage> is represented as C for Context, A for Apartment, P for Process, S for SCM, and where the <side> is represented as C for Client, R for Router, S for Server. There is no backtracking allowed between stages in a journey, e.g., the server-side apartment activators stage ("AA(S)") can't modify the prototype context and then have server-side process activators stage ("PA(S)") act on it. As shown in the Table 2, all journeys will invoke the custom activators in the server-side process activators stage ("PA(S)"), the server-side apartment activators stage ("AA(S)"), and the server-side context activators stage ("CA(S)").

Table 2. The Activation Journey

Journey		Stages Traversed
Same Context; Same Apartment; Same Process		CA(C)->PA(S)->AA(S)->CA(S)
Same Machine Different Process		CA(C)->SA(C)->SA(S)->PA(S) ->AA(S)->CA(S)
Different Machine	Normal	CA(C)->SA(C)->SA(S)->PA(S) ->AA(S)->CA(S)

Table 2. The Activation Journey	
Journey	**Stages Traversed**
LoadBalanced	CA(C)->SA(C)->SA(R)->SA(S) ->PA(S)->AA(S)->CA(S)

Activation Properties Flow. With reference to Figure 27, activation properties information flows between the activators 1352-1357 in the chain 1350 (Figure 26) in both forward and reverse directions. A current activator 1360 (Figure 27) receives a set of activation properties (called "prototype properties" 1366) that flow to the current activator from upstream activators 1362. Other activation properties (called the "as-activated properties" 1368) flow from downstream activators 1364 to the current activator 1360. Each activator 1360, 1362, 1364 can modify the activation properties to control aspects of the context an object is activated in. The current activator 1360 can delegate to a downstream activator 1364 or can complete the activation itself. The (downstream-most) activator that completes the activation must supply a set of activation properties that describe the 'as-activated' object. The as-activated properties flow on the return from each activator to its immediate upstream activator (i.e., as an [out] parameter of the activation calls). In the illustrated activation properties flow, the "as-activated" properties can be modified by the activators as the properties are flowed in the reverse direction. It is undefined whether the prototype properties passed on as delegating to a downstream activator can be used when the delegation returns with the as-activated properties. In an alternative implementation of the invention, the reverse flow can be accomplished by invoking the activation chain in a second pass that allows the activators to act on the "as-activated" properties resulting from a first pass.

With reference again to Figure 26, the activation chain starts with an immediate activator described more fully below. The immediate activator executes in the client context 1198 and is responsible for initializing all activation properties from information supplied by the client in the activation request, such as mapping between program, class and configuration identifiers (i.e., PROGID->Ref CLSID->Config ID). The immediate activator also is responsible for initializing data structures representing the prototype context and client context (hereafter termed the "prototype context representative" and the "client context representative") in the prototype properties from the policy makers in the client context 1198. In particular, the immediate activator adds each policy maker in the client context that is marked with the propagate attribute (CP_PROPAGATE) into the prototype context representative. The immediate activator also adds each policy maker of the client context that is marked with the exposed attribute (CP_EXPOSE) to the client context representative. The immediate activator further initializes an "IsClientContextOK" flag to TRUE. The client context activation state (designated "CA(C)" in Table 1 above) is now run with these activation properties.

If the server object to be activated can not be in the client's process (for example, there is a remote server name) then the "IsClientContextOK" flag is set FALSE and the client-side SCM activators stage (designated "SA(C)" in Table 1 above) is delegated. The activation properties including the prototype context, and client context representative are marshaled to the SCM. The custom activators of the client-side, router-side and server-side SCM activators stages (designated "SA(C)," "SA(R)," "SA(S)" in Table 1 above) are invoked and can modify the activation properties. Eventually, the server-side SCM activators stage ("SA(S)") will delegate to the server-side process activators stage ("PA(S)") stage and the activation properties are marshaled to that process. The custom activators 1356 in the server-side process activators stage ("PA(S)") can control activation properties affecting which apartment and context is used in its process. Eventually, the server-side process activators stage ("PA(S)") will delegate to the server-side apartment stage ("AA(S)") in the appropriate apartment. The custom activators in the server-side apartment activators stage ("AA(S)") can control activation properties affecting which context in this apartment is used. Eventually, the server-side apartment activators stage ("AA(S)") will delegate to the server-side context activators stage ("CA(S)"). The custom activators 1357 in the server-side context activators stage ("CA(S)") control what object instance is activated. If all CA(S) Custom Activators delegate then the usual COM DllCache processing is done to complete the activation. Whichever Activator that is completing the activation is responsible for initializing and returning the "As-Activated Properties".

The "IsClientContextOK" flag can be set FALSE by any custom activator in the chain 1350. It will also be set FALSE by COM+ when it determines that the Process, Apartment, or Context will change from the creator's context. Logically the server-side apartment activators stage ("AA(S)") will create a new context based upon its state.

On the other hand, if the server object 1192 can be activated in the same process as the client then the server-side process activators stage ("PA(S)") is delegated to and activation processing performed similar to the server-side SCM activators stage to server-side process activators stage ("SA(S)->PA(S)") delegation previously described.

Activation Framework. A number of assumptions apply in the above-described activation journey in the illustrated extensible object execution environment 1100 (Figure 11). One, every server object in the illustrated environment 1100 has a context, which may be a default context. Every context has an activator registered in it. Every process has a standard COM+ process activator. Every class may, optionally, have registered custom activators. Finally, there are process-wide class factory and class data tables. There is no assumption made about the activator registered on a context being specialized for that context. In the majority of contexts, the registered activator will be the standard COM+ process activator. The COM+ process activator is agile between all apartments and contexts in its process. An agile object is one which may legally be called directly from within any context or apartment in their process.

At each of the stages in the activation journey, the COM+-provided activator of the stage finds a list of any custom activators registered for the server object in the stage from the server object's class data (e.g., in the system registry or COM properties catalog). (In some alternative implementations, the client also can register custom activators to which activation is delegated in particular of the stages.) If the server object's class has a custom activator for the stage, the COM+-provided activator instantiates and initializes the custom activator, acquires the "ISystemActivator" interface from the custom activator, and delegates to the appropriate method on that interface. After the custom activator's activation processing, the custom activator will usually delegate to the COM+-provided standard activator which then invokes the next custom activator (if any) at this stage. When all custom activators have been processed, the COM+-provided activator will do its processing for the stage and either complete the activation or delegate on to the next stage for this journey. For example, when a transition from client or router-side to server-side occurs at a stage, the client or router-side custom activators are run first. The COM+-provided activator does its client or router-side processing. Then, the server-side custom activators are run, and the COM+-provided activator does its server-side activation processing.

Immediate Activators. An activation API call (e.g., the client's call to the CoCreateInstance, CoGetInstanceFromX, or CoGetClassObject API) starts activation with a COM+-provided immediate activator of the "nearest" activator stage. This immediate activator is the context activator if the call originates in a COM object, and the standard COM+ activator otherwise. The "immediate" activator goes through a sequence of steps, including identifying the class of the server object and acquiring data for the class.

Context Activators. In the client-side context activator stage, the immediate activator is run as a preliminary to executing custom activators. The immediate activator (the COM+-supplied context activator which is initially delegated activation from the object instantiation service) finds any custom activators designated for the particular server object class in the context stage and delegates activation to such custom activators. When all the custom activators have been delegated, the COM+-supplied context activator completes the activation processing in the stage as shown in Table 3.

Table 3. Stage-End Context Activator Processing.

Stage	Operation
CA(C)	If object is to be created in same process, delegate to PA(S) stage. If object is to be created in different process, delegate to SA(C) stage.
CA(S)	Obtain Class Factory: Use class table to acquire a naked reference to a class factory in this apartment. Use class factory to instantiate and possibly initialize an instance.

Table 3. Stage-End Context Activator Processing.	
Stage	**Operation**
	Query for and return the required interfaces on the new instance.

Process Activator. A COM+-supplied process activator is delegated activation from either the default context activator or the SCM activator. The process activator always receives fully resolved class identification from the initial immediate activator, and may in addition receive class data and location information. The process activator delegates to any custom activators that are designated for the particular server object class in the process, then completes activation processing for the stage as shown in Table 4.

Table 4. Stage-End Process Activator Processing.	
Stage	**Operation**
PA(S)	Determine Apartment: This process is the server process. Evaluate class properties to determine appropriate apartment for activation. Create or reuse existing Apartment. Delegate to AA(S) stage for that apartment to complete activation (freezing the choice of apartments).

Apartment Activators. A COM+-supplied apartment activator is delegated activation from the process activator as indicated in Table 3. The apartment activators are the base case for legacy COM activation (objects not in an environment extension domain). The apartment activator delegates to any custom activators that are designated for the particular server object class in the process, then completes activation processing for the stage as shown in Table 5.

Table 5. Stage-End Apartment Activator Processing.	
Stage	**Operation**
AA(S)	Context Determination: Evaluate prototype context and to locate an equivalent existing context (could be the client context) or determine that a new context is required. If the object is being activated in an existing context, delegate to the CA(S) stage for that context. If the object is being activated in a new context. Create the context, attach the default context activator, Freeze the new Context, and delegate to it.

SCM Activator. The COM+-supplied SCM activator gets control over the server object's activation as a result of delegation, either by a context activator or by a remote SCM activator. The SCM activator always receives fully resolved class identification, and may in addition receive class data and location information. The SCM activator delegates to any custom activators that are designated for the particular server object class in the process, then completes activation processing for the stage as shown in Table 6.

Table 6. Stage-End SCM Activator Processing.	
Stage	**Operation**
SA(C)	Determine Machine: If location information indicates another machine, then delegate to SA(S) stage or SA(R) on the remote machine. If location says 'this machine' start stage SA(S) on this machine.
SA(R)	Determine Machine: If location information indicates another machine, then delegate to SA(S) stage on the remote machine. If location says 'this machine' start stage SA(S) on this machine.
SA(S)	Determine Process: This machine is where this activation will occur. Select or create server process. Delegate to PA(S) stage in that process (this freezes the choice of processes).

Class Factory Client Context Flow. In accordance with COM, the client object 1190 also can initiate activation of the server object by calling the "IClassFactory::CreateInstance()" method on the class factory 222 (Figure 4) of the server object. In such case, it is necessary to also flow the client context information through the activator chain, as with activation through an object instantiation API. Otherwise, the most likely result is that an instance is created and wrapped in the same context as the context of the class factory. In the illustrated environment 1100, this is

accomplished by substituting a reference to a helper object for that of the class factory in response to a request to the "CoGetClassObject" API (which is used to request a class factory reference). The helper object provides an implementation of the "IClassFactory::CreateInstance()" method that uses the "CoCreateInstance" API or an immediate activator's CreateInstance method to activate the server object with the server object's actual class factory. This ensures that the client context information flows on the client's "IClassFactory::CreateInstance" request and that the COM activation semantic, "CoGetClassObject()->CreateInstance," is identical to a "CoCreateInstance()" API call.

Activation Interfaces. Figures 28 and 29 show program listings 1370, 1372 of interfaces used in the activators chain just described. The "CoGetObjectContext()" API (Figure 28) is a service provided in the COM+ runtime services that returns an interface pointer on an object's object context object. The standard activators for the stages in the activation journey implement the "IActivator" interface (Figure 28), which includes "CreateInstance()" and "GetClassObject()" methods. The "CreateInstance()" method creates an instance of a specified server object, and returns an interface pointer on the server object. The "GetClassObject()" method is used to obtain an interface on a class factory (e.g., the class factory 222 of Figure 4) of a specified server object. The "ISystemActivator" interface (Figure 28) is implemented by custom activators to which the system-provided standard activators delegate during activation processing in the various stages of the activation stage as described above. The "ISystemActivator" interface likewise includes "CreateInstance()" and "GetClassObject()" methods. The methods in the "ISystemActivator" interface include parameters that pass the above-described prototype properties downstream, and as-activated properties upstream through the activation chain. The activation properties are initialized and manipulated using the "IInitActivationPropertiesIn," "IActivationPropertiesIn," and "IActivationPropertiesOut" interfaces shown in Figure 29.

Having described and illustrated the principles of our invention with reference to an illustrated embodiment, it will be recognized that the illustrated embodiment can be modified in arrangement and detail without departing from such principles. It should be understood that the programs, processes, or methods described herein are not related or limited to any particular type of computer apparatus, unless indicated otherwise. Various types of general purpose or specialized computer apparatus may be used with or perform operations in accordance with the teachings described herein. Elements of the illustrated embodiment shown in software may be implemented in hardware and vice versa.

In view of the many possible embodiments to which the principles of our invention may be applied, it should be recognized that the detailed embodiments are illustrative only and should not be taken as limiting the scope of our invention. Rather, we claim as our invention all such embodiments as may come within the scope and spirit of the following claims and equivalents thereto.

CLAIMS

We Claim:

1. A computer-implemented method for providing an architecture accommodating load balanced instantiation of software objects within a server group of a plurality of server computers, the architecture being transparent to the objects, the method comprising the steps:

interposing a wrapper between a first load balanced object instantiated at a host server computer of the server group and a client of the first load balanced object, the wrapper intercepting calls to the first load balanced object to generate a performance measurement at the host server computer transparently to the first load balanced object;

at a router computer, monitoring performance at the server computers in the server group by collecting the performance measurement from the host server computer and at least one other performance measurement from other server computers in the server group to ascertain measured performance at the servers; and

when measured performance of the host server is preferred over measured performance of other server computers in the server group, instantiating a second load balanced object on the host server computer as a result of the performance measurement.

2. A computer-readable medium having computer-executable instructions for performing the steps of claim 1.

3. The method of claim 1 wherein

the first object is of an object class;

the performance measurement is identified by object class as a measurement of performance for objects having the object class of the first object;

the monitoring step determines performance at server computers in the server group for objects having the object class of the first object; and

the instantiating step instantiates the second object at the host server computer based on whether the second object has an object class matching the object class of the first object.

4. The method of claim 1 wherein a plurality of objects of a same specified object class are instantiated on a selected target group associated with the object class, the target group being a subset of the server group of server computers.

5. The method of claim 1 wherein a selected replaceable load balancing engine designated for an object class determines when monitored performance of the host server computer is preferred over monitored performance of other server computers in the server group.

6. The method of claim 1 wherein the monitoring step comprises periodically blending the performance measurement with plural other similar performance measurements from the host server computer to generate a performance value indicative of performance at the host server computer, the blending sufficiently delayed by a time interval to dampen variance in the performance value.

7. The method of claim 6 wherein the time interval is selected by a system administrator.

8. The method of claim 1 wherein the performance measurement is generated transparently to the first object by a server-side policy of the first object, wherein the policy is invoked by the wrapper.

9. The method of claim 8 wherein

the server-side policy is shared by a third object at the server computer; and

the server-side policy generates an additional performance measurement associated with the third object, wherein the additional performance measurement is collected by the router computer to monitor host server computer performance.

10. The method of claim 8 wherein the performance measurement indicates method response time at the host server computer as generated by the server-side policy of the first object, the policy operative to observe a first time as a result of an enter event sent to the policy before a method call to the object, the policy further operative to observe a second time as a result of a leave event sent to the policy after the method call to the object, the policy further operative to generate the performance measurement by subtracting the first time from the second time.

11. The method of claim 8 wherein the performance measurement indicates database load as observed at the host server computer transparently to the first object by code invoked by the wrapper.

12. The method of claim 8 wherein the performance measurement indicates database access latency as observed at the host server computer transparently to the first object by code invoked by the wrapper.

13. In a computer network, a load balancing service for distributing object related resource requests among a plurality of load bearing targets identified by target identifiers, the load balancing service comprising:

a routing table mapping object class identifiers to target identifiers, at least one of the mapped object class identifiers mapped to a target identifier identifying a target exhibiting a favorable processing metric for processing objects having an object class of the mapped object class identifier; and

a system service for receiving an object related resource request comprising a supplied object class identifier matching an object class identifier in the table, the system service operative to route the resource request to the target identified by the target identifier mapped to by the supplied class identifier in the routing table.

14. The service of claim 13 wherein the system service is an operating system service.

15. The service of claim 13 wherein the target identifiers in the routing table identify server computers in the computer network.

16. The service of claim 13 wherein target identifiers in the routing table identify threads at server computers in the computer network.

17. The service of claim 13 wherein the target identifiers in the routing table identify processes at server computers in the computer network.

18. The load balancing service of claim 13 further comprising:

a collective metric data store operative to receive performance values indicative of the targets' performance according to a processing metric; and

a load balancing engine operative to dynamically adjust mappings in the routing table to map an object class identifier to a target identifier identifying a target having a more favorable performance value in the collective metric data store according to the processing metric.

19. The load balancing service of claim 18 wherein

the collective metric data store comprises a dampened performance value indicative of a plurality of blended received performance values, at least certain of the blended received performance values of sufficient age to dampen variance in the dampened performance value in the collective metric data store; and

the load balancing engine consults the dampened performance value to dynamically adjust mappings in the routing table.

20. In a router computer, a load balancing service for distributing an object creation request comprising an object creation request characteristic among a target group comprising a plurality of server computers, the load balancing service comprising:

a host server computer selection means for selecting a host server computer to host the object creation request, the host server computer selection means operative to accept a supplied object creation request characteristic and specify a selected host server computer, the specified host server computer selected by the host server computer selection means from the target group based on the specified object creation request characteristic;

a system service for routing the component creation request to a server computer in the target group, the system service operative to accept the component creation request comprising the object creation request characteristic, supply the object creation request characteristic to the host server computer selection means, accept from the host server computer selection means a specified selected host server computer and route the object creation request thereto.

21. The load balancing service of claim 20 wherein the object creation request characteristic of the object creation request is a class identifier, and the host server computer selection means specifies a server computer having an activated instantiated object of the class identifier.

22. The load balancing service of claim 20 wherein the object creation request characteristic of the object creation request is a client identity identifying a client computer, and the host server computer selection means specifies a server computer to which object creation requests for the client computer have previously been routed based on the previously-routed requests.

23. The load balancing service of claim 20 wherein the object creation request characteristic is selected from the group consisting of a class identifier of the object creation request, a client computer identity of a computer issuing the object creation request, and a process identity of a process issuing the object creation request.

24. The load balancing service of claim 20 wherein the object creation request is generated by a process executing in an execution environment with associated environment data defining the execution environment, and the object creation request characteristic of the object creation request is the execution environment data.

25. The load balancing service of claim 20 wherein the target group is dynamically adjustable to add or remove a specified server computer from the group based on instructions provided by the specified server computer to the router computer.

26. The load balancing service of claim 20 wherein the object creation request characteristic of the object creation request is a class identifier, the load balancing service further comprising:

a global routing table mapping object class identifiers to host server computers in the target group,

wherein the host server computer selection means consists of a mapping function on the global routing table.

27. The load balancing service of claim 26 wherein the global routing table is dynamically updated with processing metric data retrieved from the server computers in the target group.

28. The load balancing service of claim 27 wherein the processing metric data is generated by a RPC layer between the router computer and the server computers.

29. In a computer network, an object creation architecture for balancing a load of object creation requests among a plurality of server computers, the architecture comprising:

a routing table comprising a plurality of stored object class identifiers, wherein at least one stored object class identifier is associated with a server computer;

at a router computer, a load balancing service responsive to a supplied object class identifier in an object creation request from a client program on a client computer and operative to select a server associated with the supplied object class identifier in the routing table, the load balancing service further operative to route the object creation request to an object creation service at the selected server computer;

at the selected server computer, an object creation service responsive to the object creation request from the load balancing service and operative to create a server object of an object class associated with the supplied identifier and further operative to assemble a stub with the server object, the stub operative to monitor calls to the server object to observe and store in a metric data store at the selected server computer a performance value, the performance value indicative of performance at the selected server computer according to a processing metric;

at the router computer, a metric collector operative to retrieve the observed performance value from the metric data store and integrate the performance value into a collective metric data store, wherein the collective metric data store comprises metric data from plural server computers; and

a load balancing engine at the router computer operative to consult the collective metric data store and associate in the routing table an object class identifier with a server having a

performance value determined superior according to the processing metric by the load balancing engine.

30. The architecture of claim 29 wherein the load balancing engine is a first load balancing engine operative to associate a first class identifier with a server, the architecture further comprising:

a second load balancing engine operative to consult the collective metric data store and associate in the routing table a second class identifier with a server having a performance value determined superior according to the processing metric by the second load balancing engine.

31. In a computer network comprising a router computer, a plurality of server computers and a plurality of client computers, an architecture for balancing a load of computer object processing among the server computers, the architecture comprising:

a routing table at the router computer associating object classes with server computers;

a monitor at a server computer, the monitor operative to intercept a reference to an instantiated first software object of a monitored object class to transparently conduct and record a processing metric observation, the monitor further operative to send a processing metric value based on the processing metric observation and indicative of performance at the server computer;

a load balancing service at the router computer, the load balancing service operative to receive a client computer request to create a second object of the monitored object class and route the request to a selected server associated with the monitored object class in the routing table, the load balancing service responsive to the processing metric value sent by the monitor to associate a server having a favorable processing metric value with the monitored object class in the routing table; and

an object creation service at the selected server operative to receive the request from the load balancing service and create an object of the monitored object class.

32. The architecture of claim 31 wherein the load balancing service comprises a plurality of load balancing engines, each load balancing engine operative to modify associations in the routing table relating to an object class particular to the load balancing engine.

33. The architecture of claim 31 wherein the monitor comprises a plurality of metric trackers, each metric tracker operative to conduct and record a performance metric observation relating to an object class particular to the metric tracker.

34. In a computer network having a router computer and a plurality of server computers in a target group, a method for balancing object processing among the plurality of server computers, the method comprising:

conducting plural processing performance metric observations associated with a software object class at a server computer;

periodically blending the observations into a representative value indicative of performance at the server computer;

periodically transferring the representative value from the server computer to a router computer to provide plural successive representative values to the router computer, wherein transferring is sufficiently delayed to facilitate blending a number of observations to dampen variance in the successive representative values;

receiving at a router computer the plural representative values from the server computer and plural representative values from at least one other server computer in the target group; and

routing resource requests received by the router computer to a server computer in the target group having a representative value indicative of more favorable performance than another server computer in the target group.

35. The method of claim 34 wherein the blending step comprises calculating the average of plural response times observed at the server computer.

36. In a computer network comprising a client computer and a server computer, a computer implemented method for accommodating object transparent rebalance messages from the server computer to the client computer, the method comprising:

generating a proxy at the client computer for receiving and forwarding calls from a client program to a software object;

generating a stub at the server computer for receiving and forwarding calls from the proxy to the software object;

establishing a connection between the proxy and the stub;

providing the proxy with rebalance message receptive code run by the proxy transparently to the server object and the client program;

providing the stub with rebalance message generating code run by the stub transparently to the server object and the client program;

when the stub is referenced, determining in the rebalance message generating code whether performance at the server computer is below an acceptable minimum; and

if performance at the server computer is below the acceptable minimum, sending a rebalance message from the rebalance message generating code to the rebalance message receptive code.

37. The method of claim 36 wherein the software object is a first software object, the method further comprising:

upon receiving a rebalance message in the rebalance receptive code, severing the connection between the proxy and the stub and creating a second software object on a computer other than the server computer.

38. A load balancing service for balancing object processing among a plurality of server computers by accommodating object creation requests from a plurality of client programs executing on a plurality of client computers, the load balancing service comprising:

at a client computer, a configuration database for associating object classes with remote computers, at least one object class in the configuration database associated with a router computer;

at the client computer, an operating system service operative to receive an object creation request comprising an object class, the operating system service further operative to direct the object creation request to a computer associated with the object class in the configuration database;

at the router computer, a routing table for associating an object class with a server computer;

at the router computer, a routing service operative to receive the object creation request from the client computer and route the request to a selected server computer associated with the request's object class in the routing table;

at the selected server computer, a class instance creator operative to receive the object creation request and create an object of the request's object class;

an operating system service for providing a wrapper around the object, the wrapper comprising a method invocation service, the method invocation service operative to receive invocations of a method of the object from a client program and forward the invocation to the object, the method invocation service further operative to observe execution of the method by the object to produce an object class method performance observation, the object class method performance observation associated with the object class of the object and indicative of the method's performance according to a processing metric;

an observation collection service at the router computer operative to collect and store the object class method performance observation and at least one other object class method performance observation in a collective observation store from a target group, the target group comprising the selected server computer and other plural server computers; and

a load balancing engine at the router computer operative to evaluate the object class method performance observations from the target group to associate a favorable server computer in the target group with a selected object class in the routing table, the evaluated observations associated with the selected object class in the collective observation store, the favorable server

computer having a more favorable object class method performance observation than another server in the target group according to the processing metric.

39. A computer-readable medium having stored thereon a data structure for routing object creation requests from a remote client computer, the data structure comprising:

identifiers indicative of an object class; and

a server identifier associated in the data structure with a selected one of the identifiers indicative of an object class, the server identifier indicative of a server computer providing to a router computer favorable processor metric observations for processing objects of the object class indicated by the selected one of the identifiers indicative of an object class.

40. The computer-readable medium of claim 39 wherein the processing metric is response time for a method of an object of the object class indicated by the selected identifier indicative of an object class, and the data structure is sharable among a plurality of load balancing engines at the router computer and an activator at the router computer.

41. The computer-readable medium of claim 39 wherein the data structure is an accelerated routing table comprising a lookup table associating a server identifier with a hash function of an identifier indicative of an object class.

FIG. 1

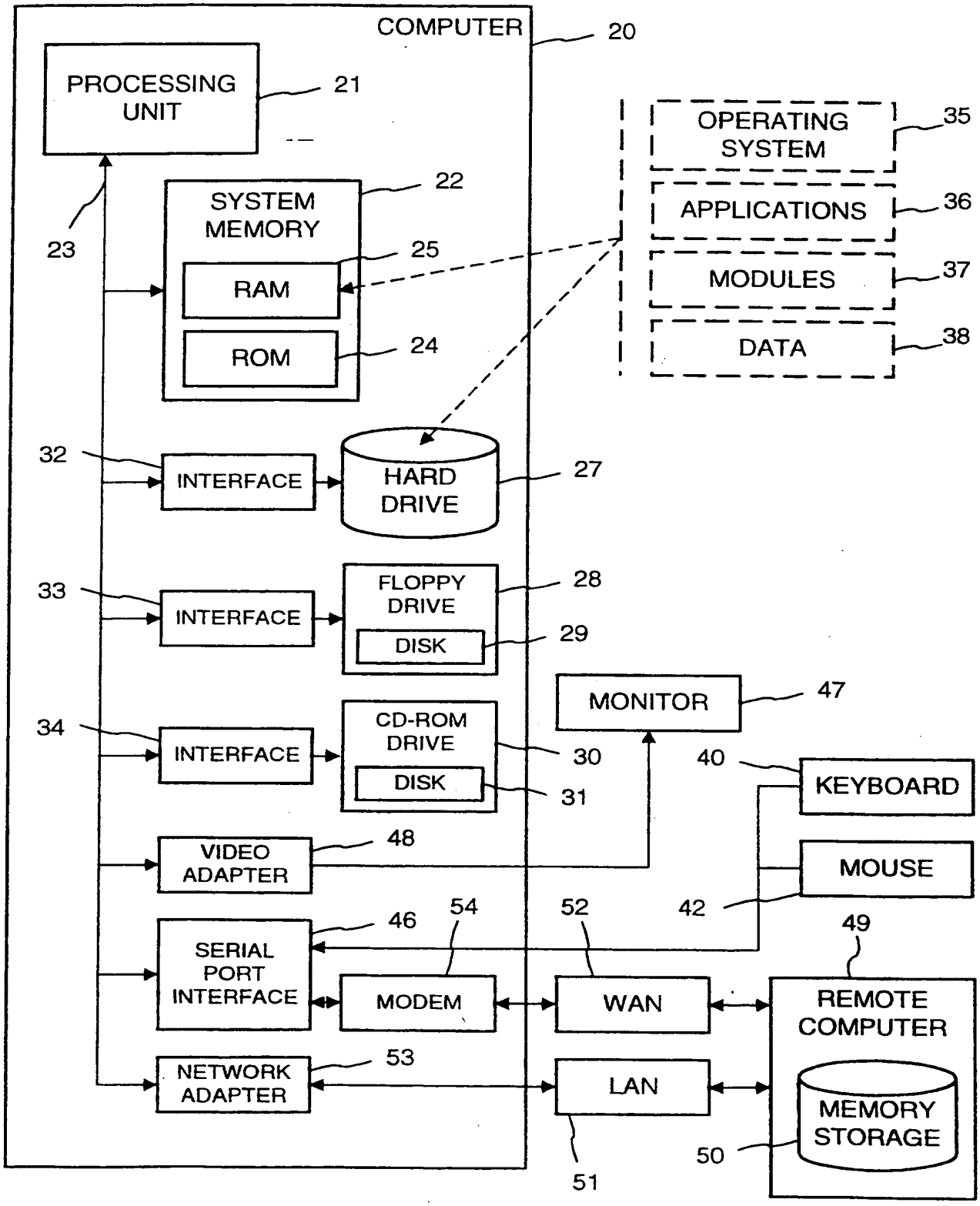


FIG. 2

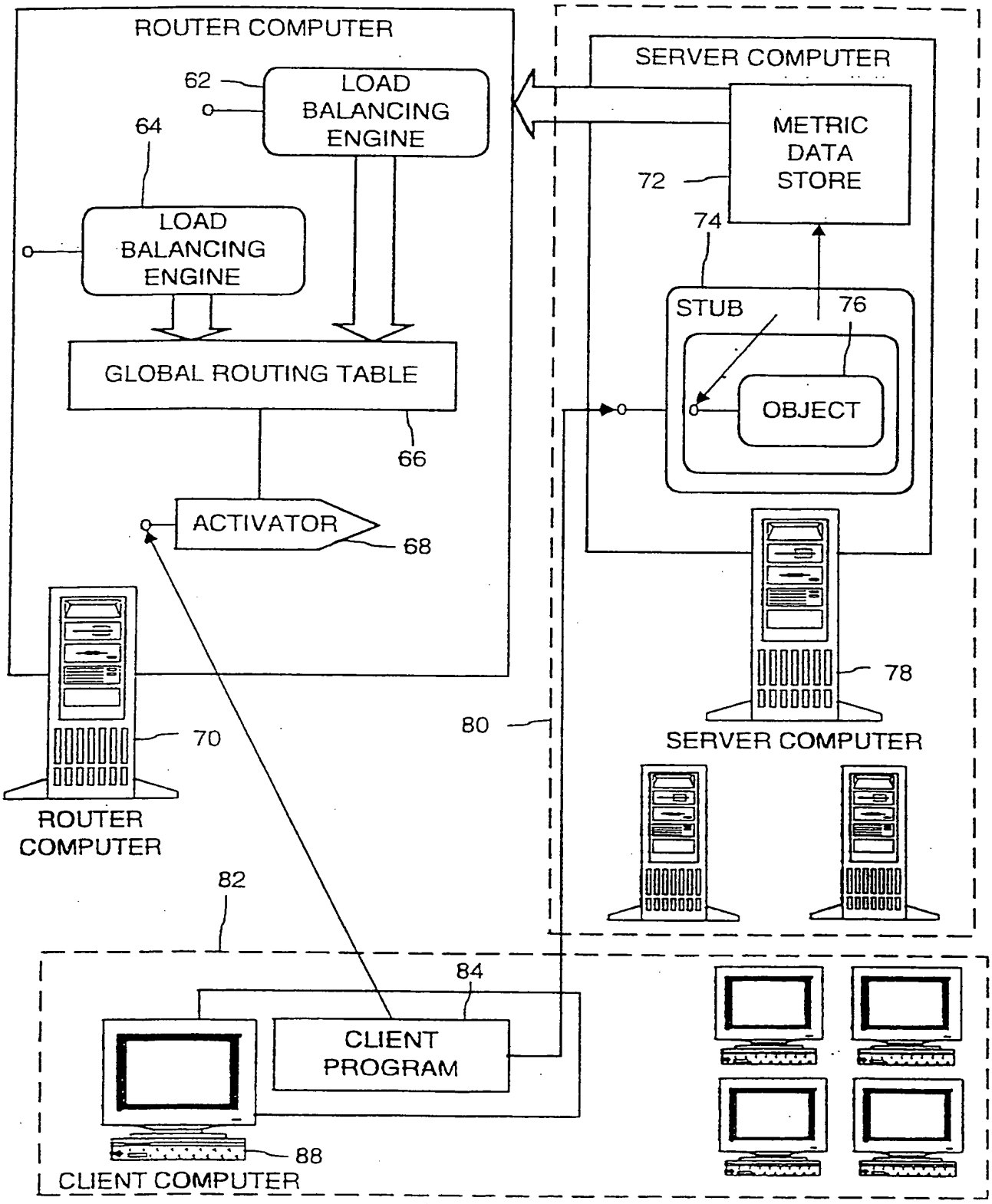

ROUTER COMPUTER
62
LOAD BALANCING ENGINE
64
LOAD BALANCING ENGINE
GLOBAL ROUTING TABLE
66
ACTIVATOR
68
70
ROUTER COMPUTER
SERVER COMPUTER
METRIC DATA STORE
72
74
STUB
76
OBJECT
78
SERVER COMPUTER
80
82
84
CLIENT PROGRAM
88
CLIENT COMPUTER

FIG. 3

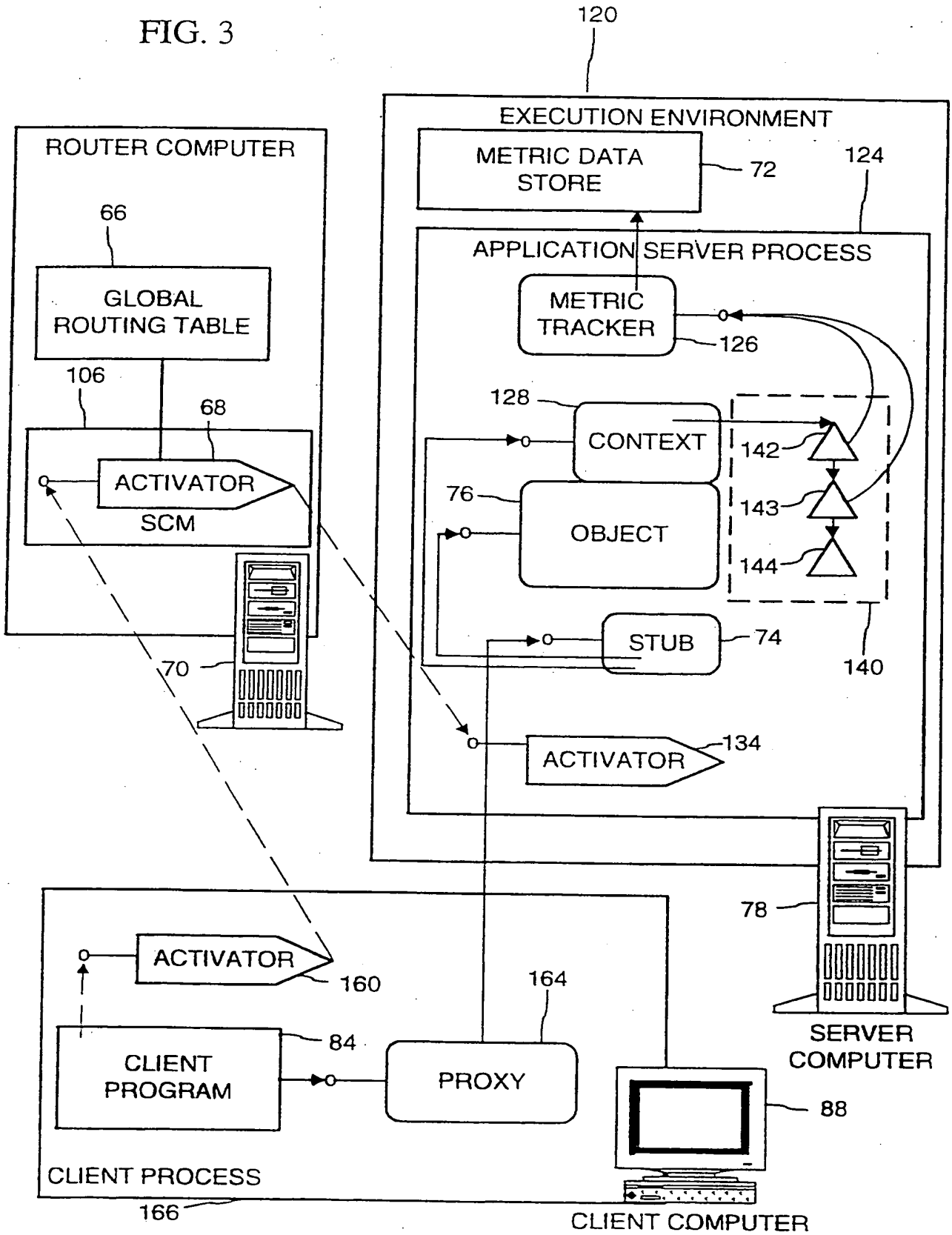


FIG. 4

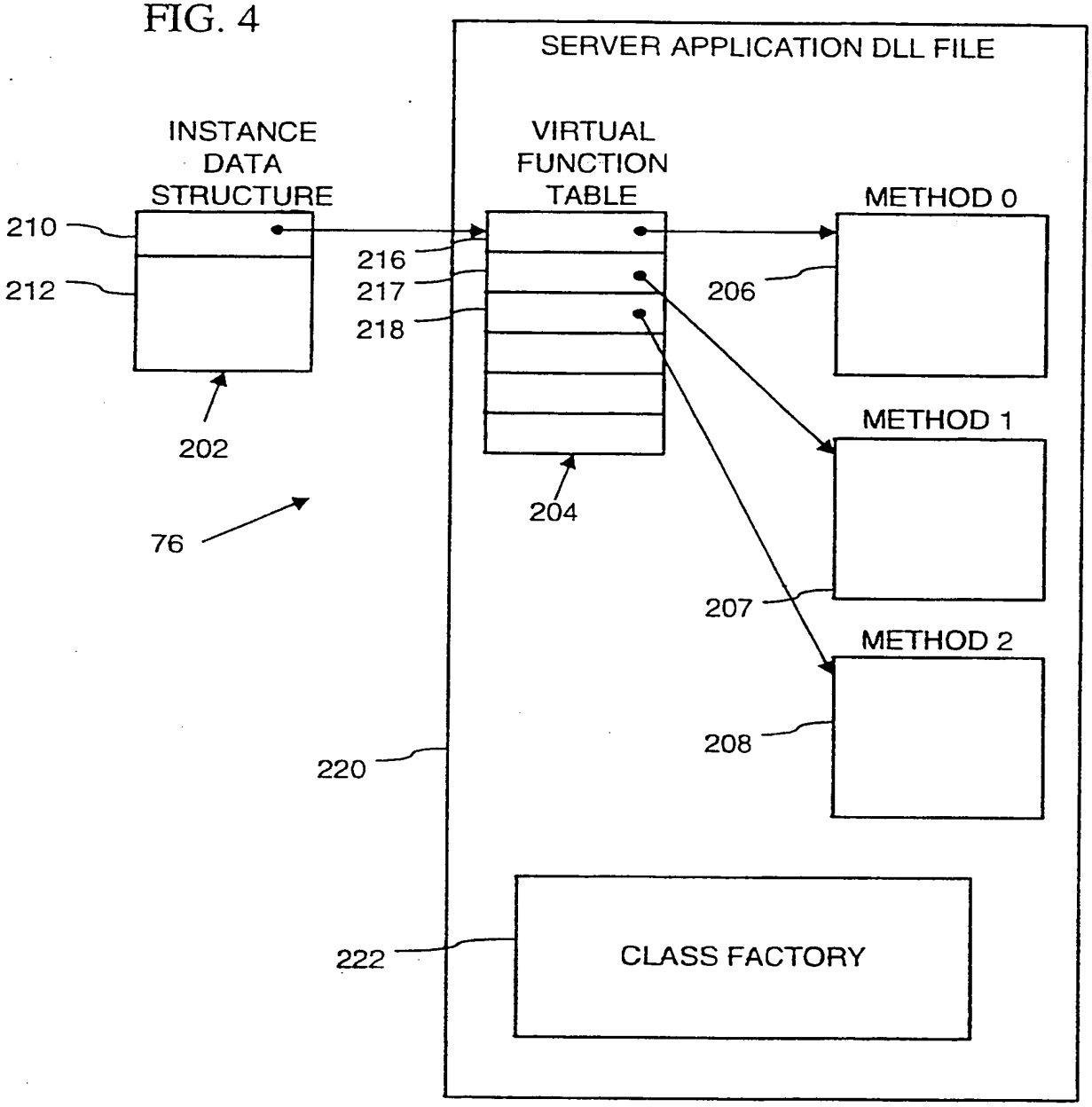

SERVER APPLICATION DLL FILE
INSTANCE DATA STRUCTURE
VIRTUAL FUNCTION TABLE
METHOD 0
METHOD 1
METHOD 2
CLASS FACTORY
210
212
202
76
216
217
218
204
206
207
208
220
222

FIG. 5

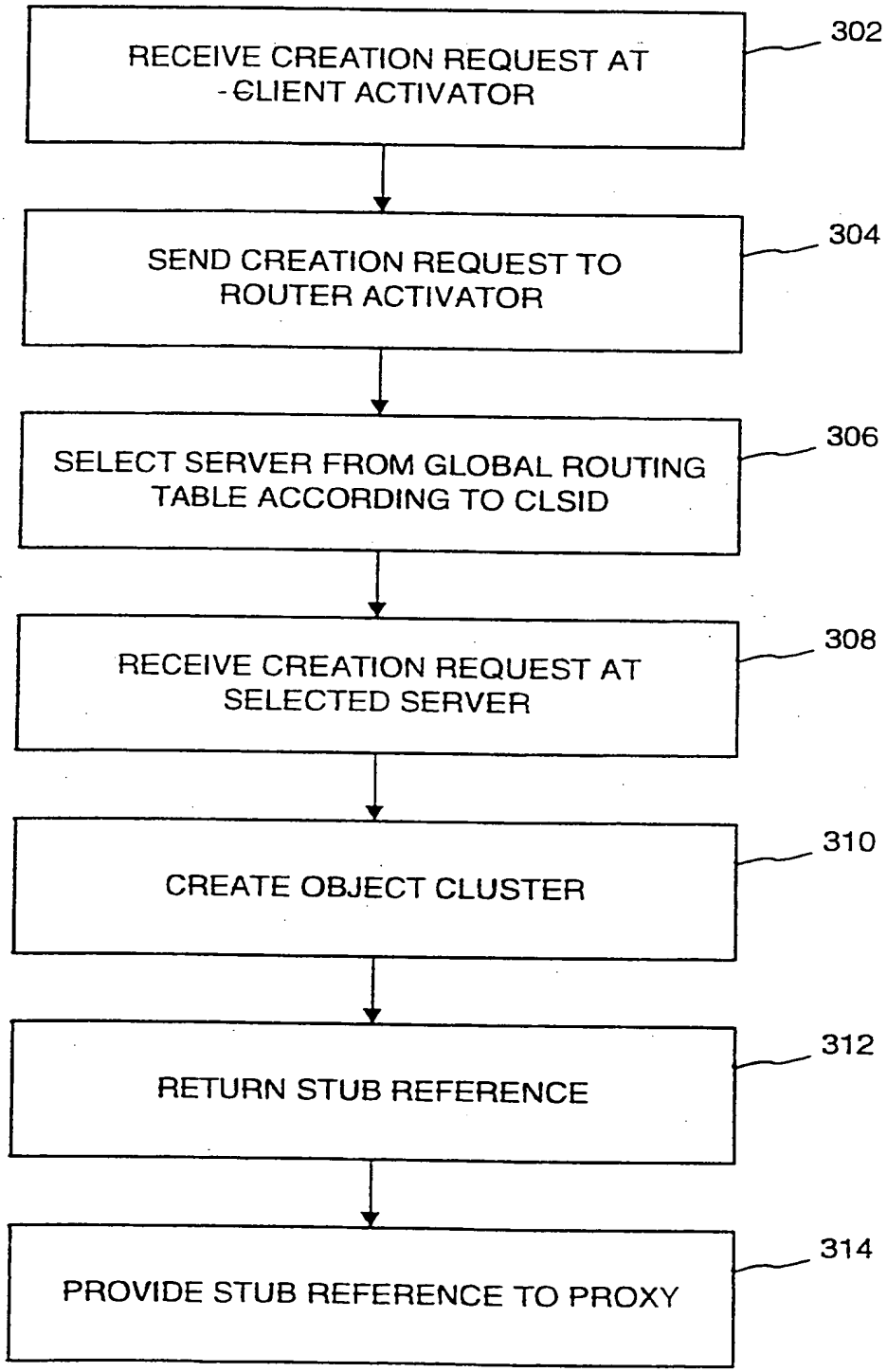


FIG. 6

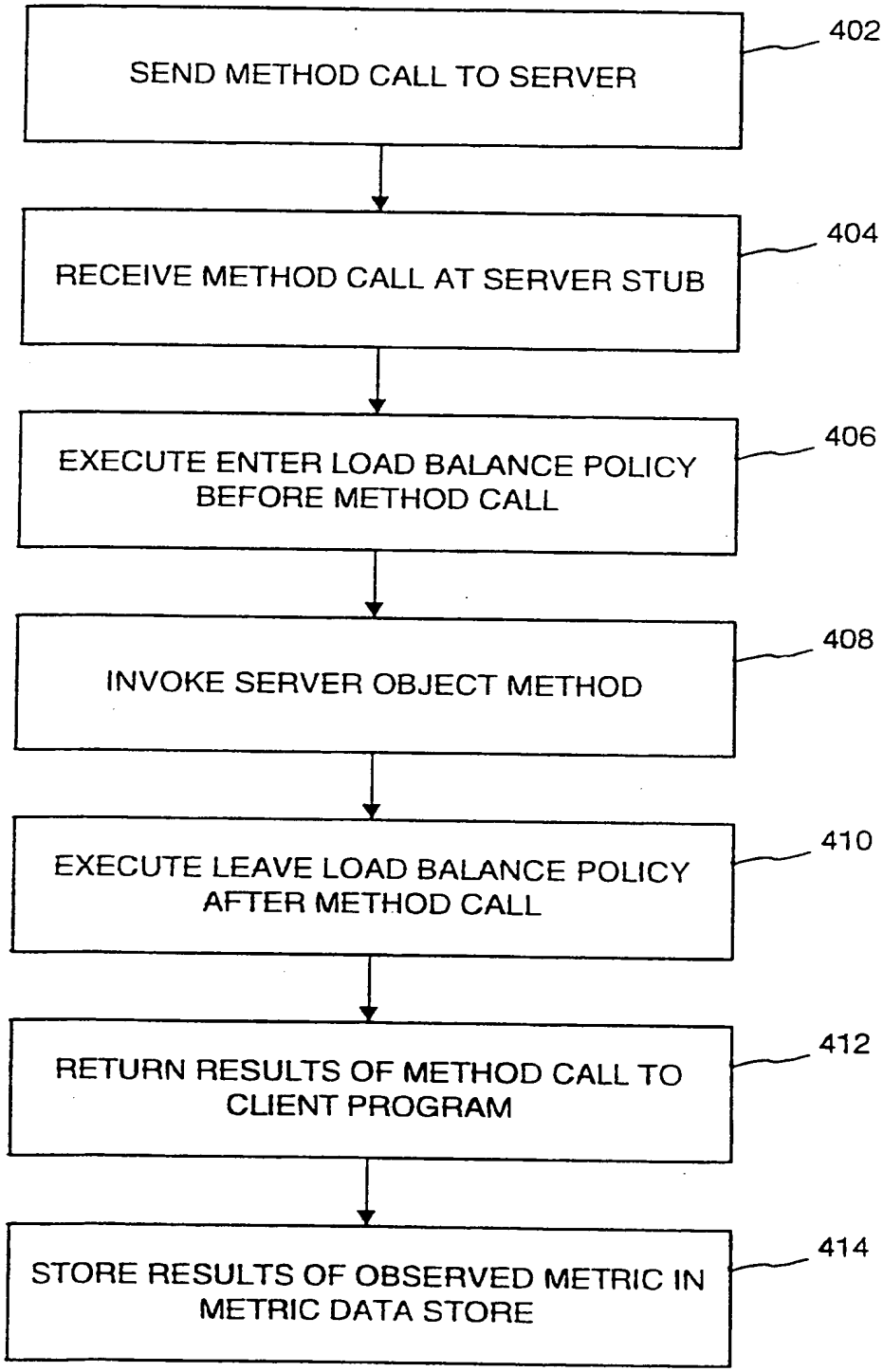


FIG. 7

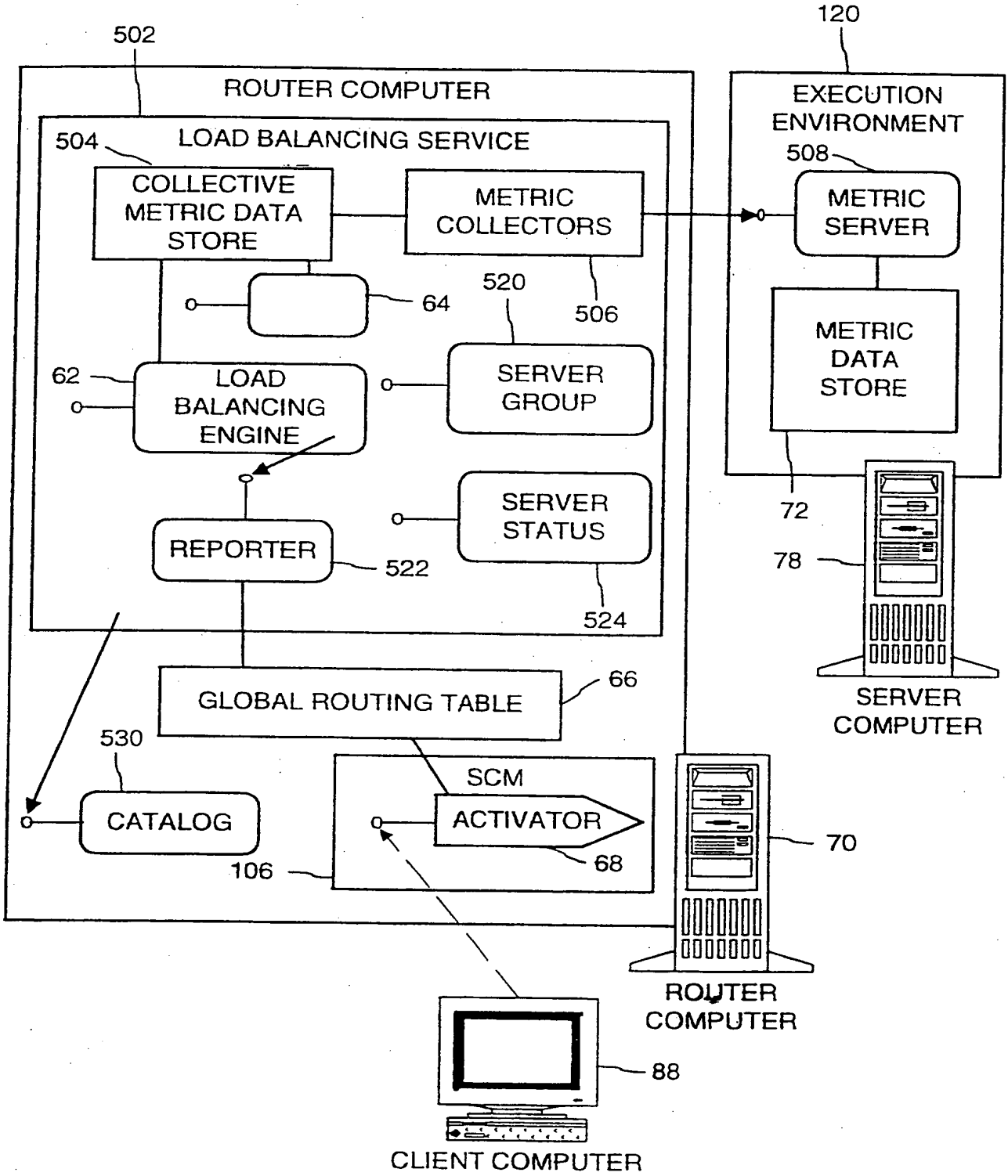


FIG. 8

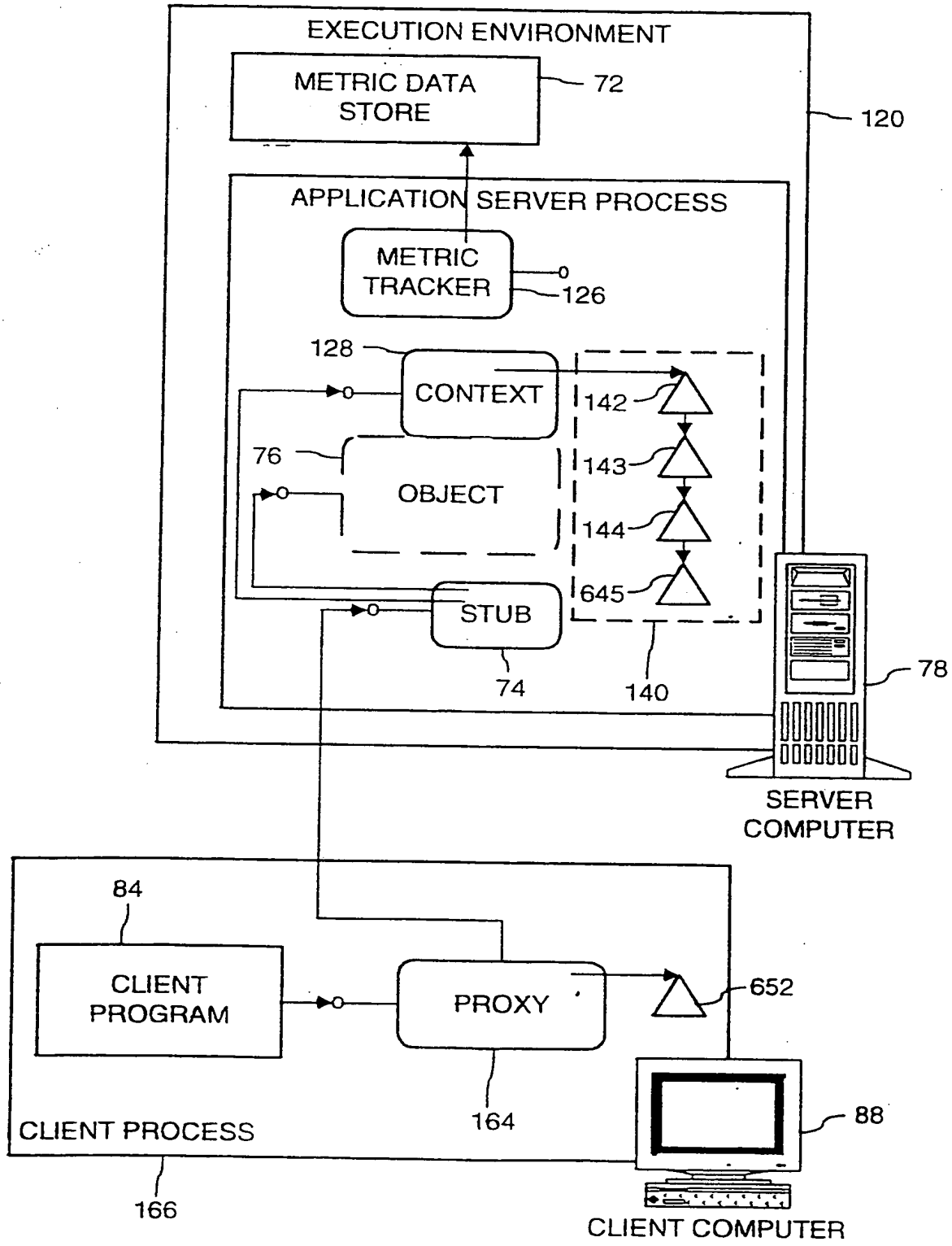


FIG. 9

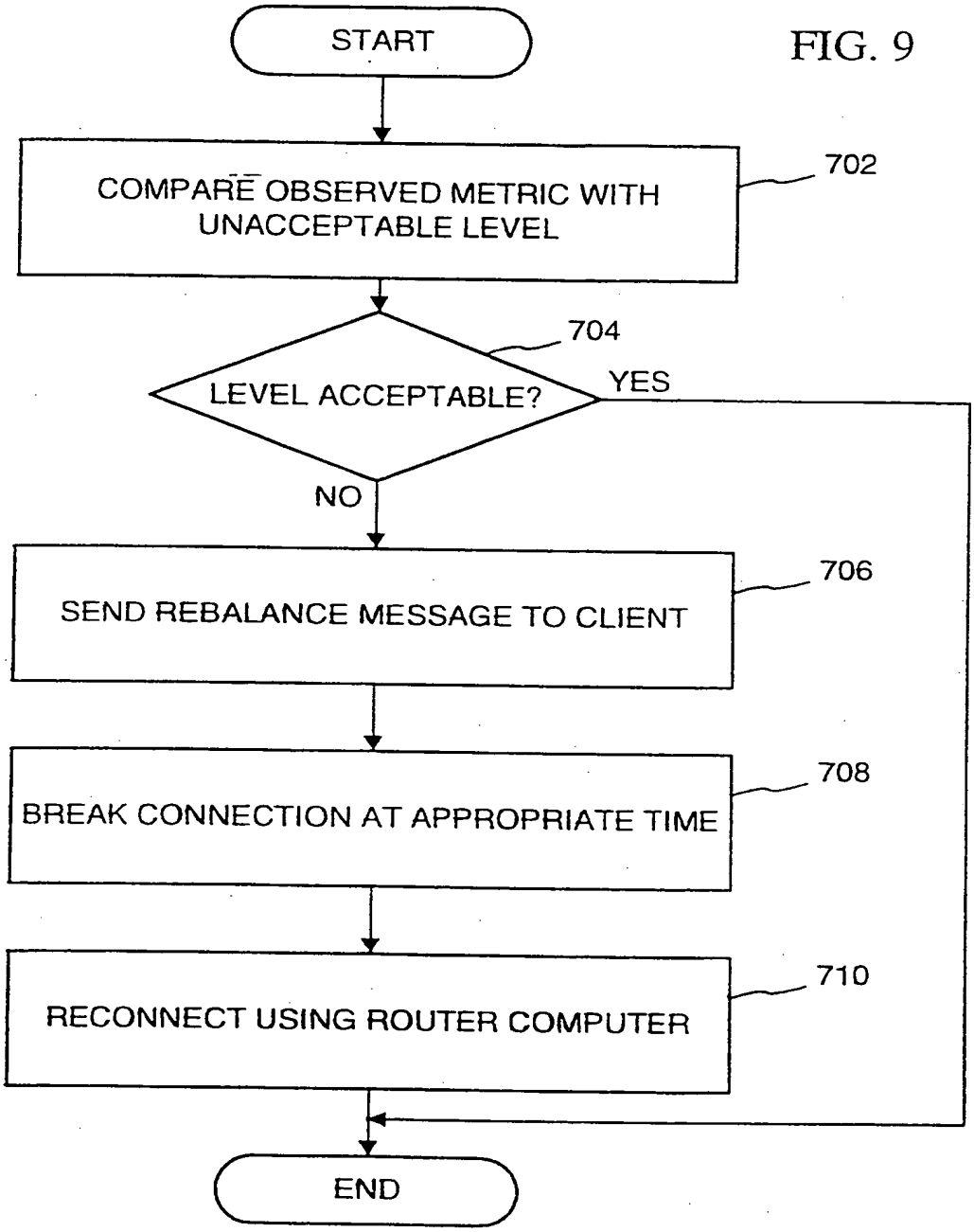


FIG. 10

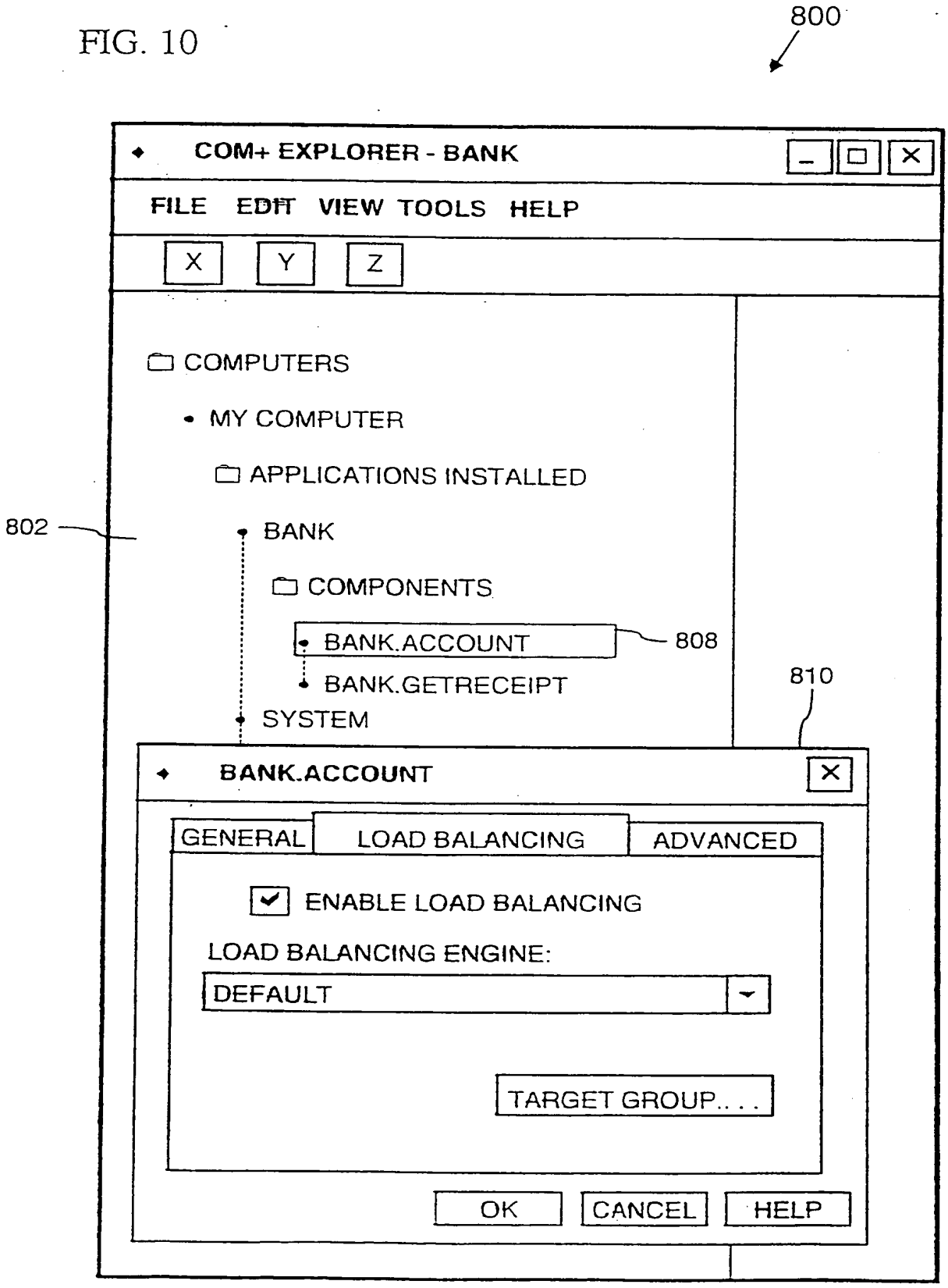


FIG. 11

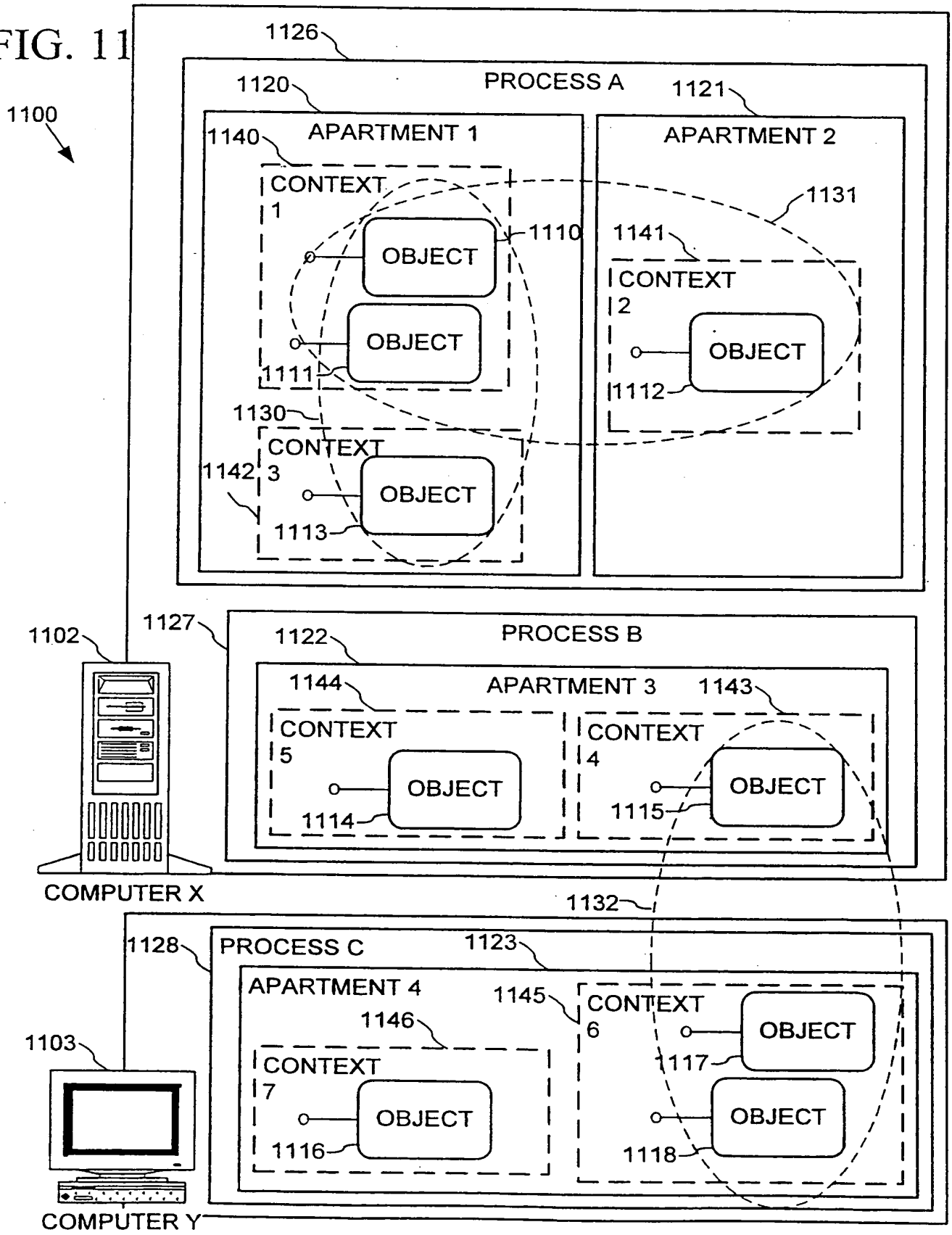


FIG. 12

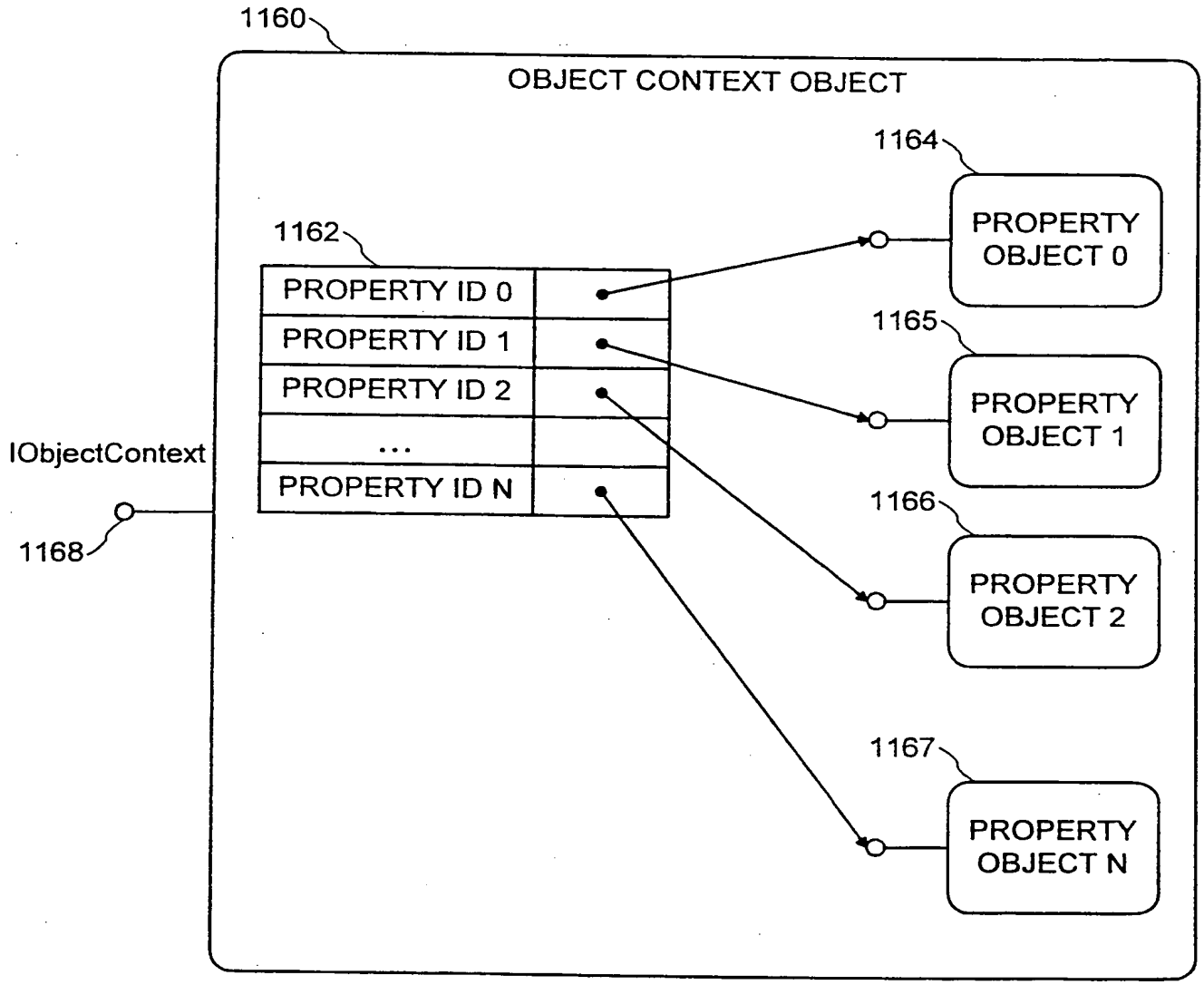


FIG. 13

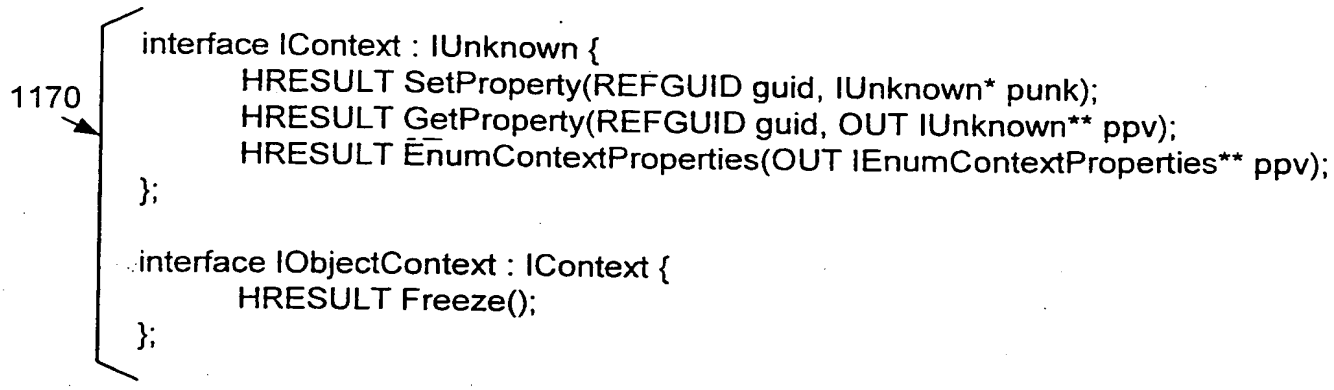


FIG. 14

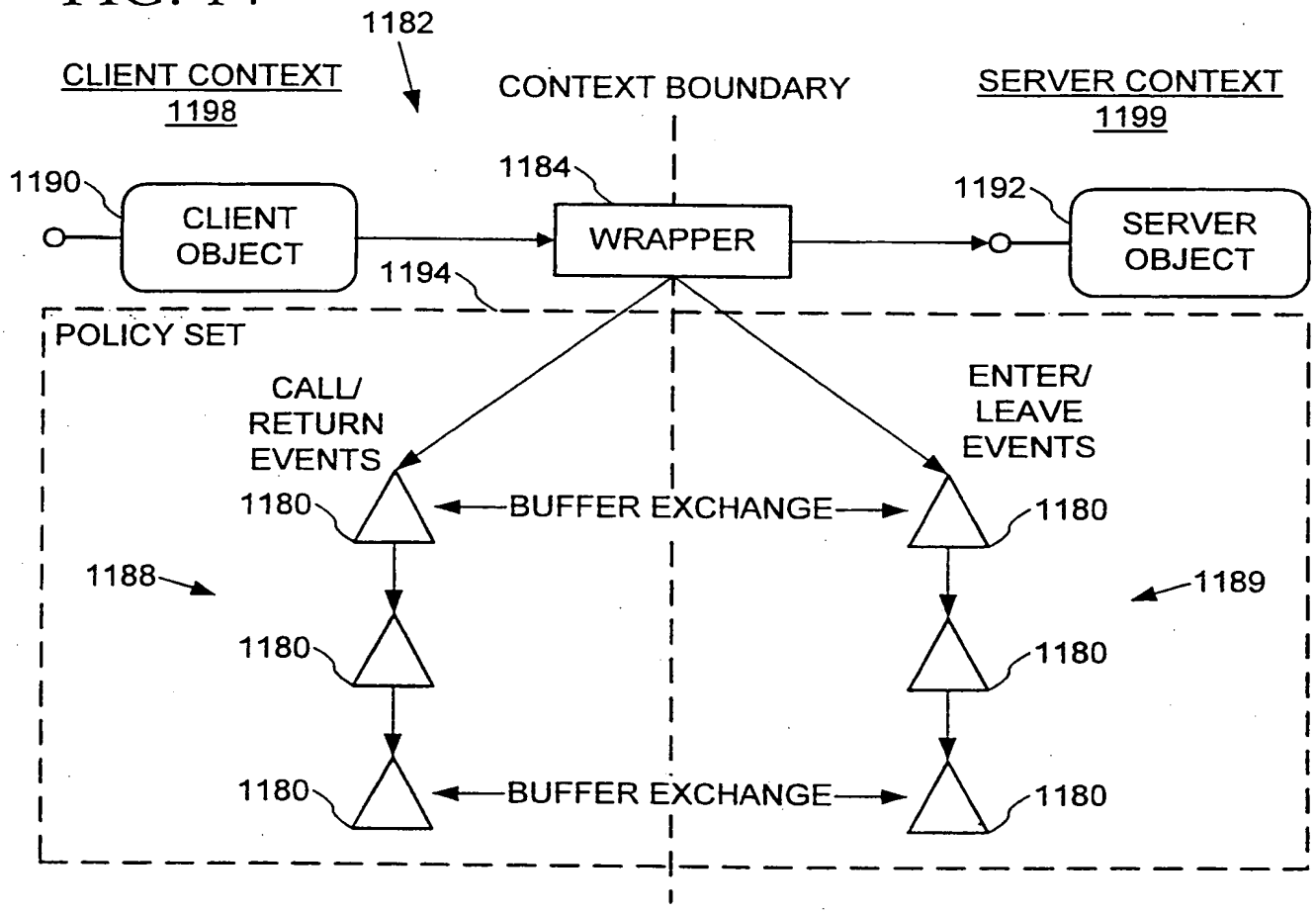


FIG. 15

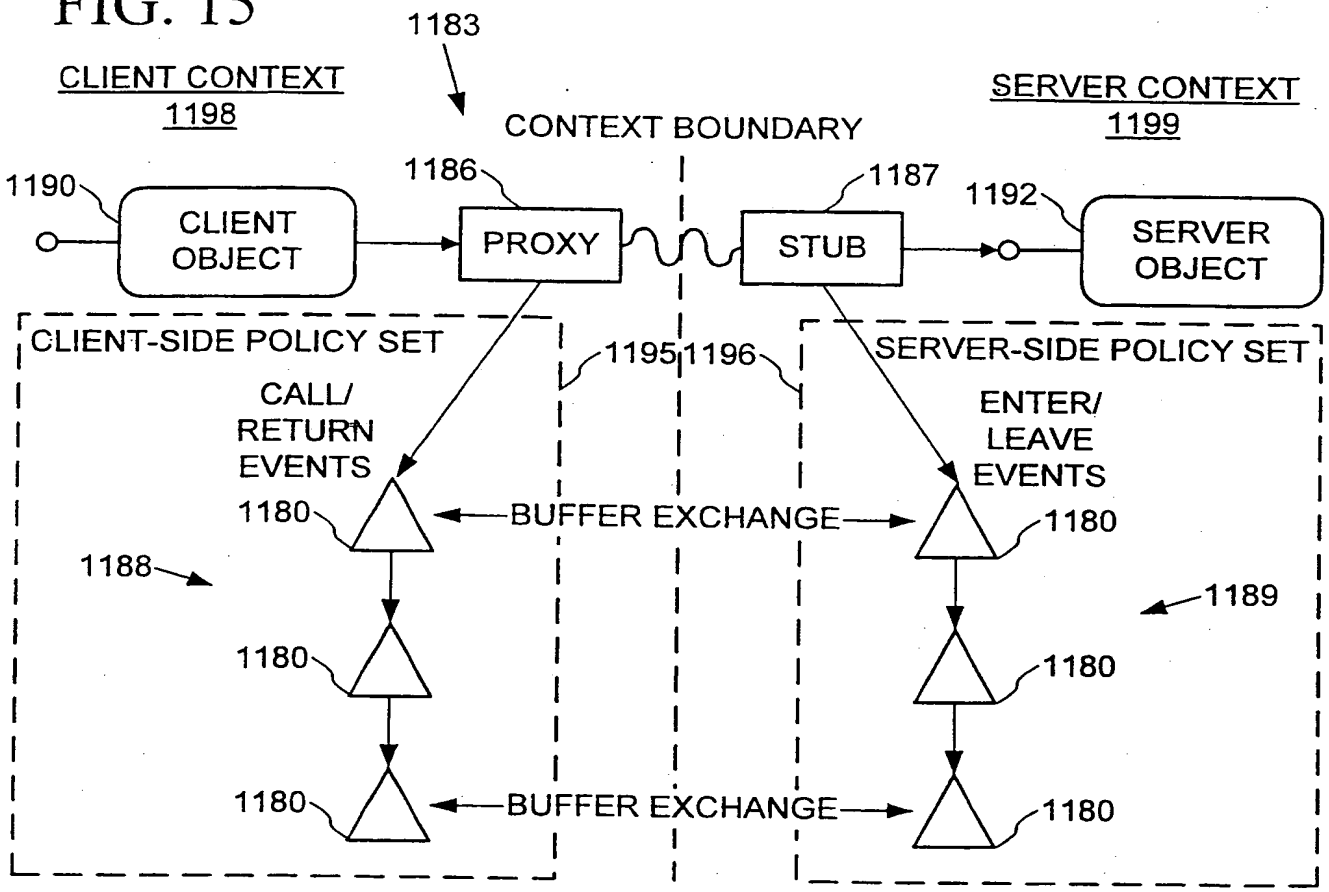


FIG. 16

1181

```
typedef ULONG ContextEvent;
const ContextEvent =
        ceCall = 1, ceReturn = 2, ceEnter = 4, ceLeave = 8,
        ceCallGetSize = 0x100, ceCallFillBuffer = 0x200,
        ceEnterWithBuffer = 0x400, ceLeaveGetSize = 0x1000,
        ceLeaveFillBuffer = 0x2000, ceReturnWithBuffer = 0x4000,
        ceReleasePolicy = 0x10000, ceAddRefPolicy = 0x20000;
interface IPolicy : IUnknown {
        // New events
        HRESULT Signal(ULONG ceType, IRpcCall *pcall);

        // Call events
        HRESULT Call(IRpcCall* pcall);
        HRESULT Enter(IRpcCall* pcall);
        HRESULT Leave(IRpcCall* pcall);
        HRESULT Return(IRpcCall* pcall);

        // Call events which pass a buffer
        HRESULT CallGetSize(IRpcCall* pcall, ULONG* pcb);
        HRESULT CallFillBuffer(IRpcCall* pcall, ULONG* pcb, void* pvBuf);
        HRESULT EnterWithBuffer(IRpcCall* pcall, ULONG cb, void* pvBuf);
        HRESULT LeaveGetSize(IRpcCall* pcall, OUT ULONG* pcb);
        HRESULT LeaveFillBuffer(IRpcCall* pcall, ULONG* pcb, void* pvBuf);
        HRESULT ReturnWithBuffer(IRpcCall* pcall, ULONG cb, void* pvBuf);

        // Policy added or released
        HRESULT ReleasePolicy(IRpcCall* pcall);
        HRESULT AddRefPolicy(IRpcCall* pcall);
};

interface IRpcCall : IUnknown {
        HRESULT GetParameter(void **);
        HRESULT nullify(HRESULT);
}
```

FIG. 17

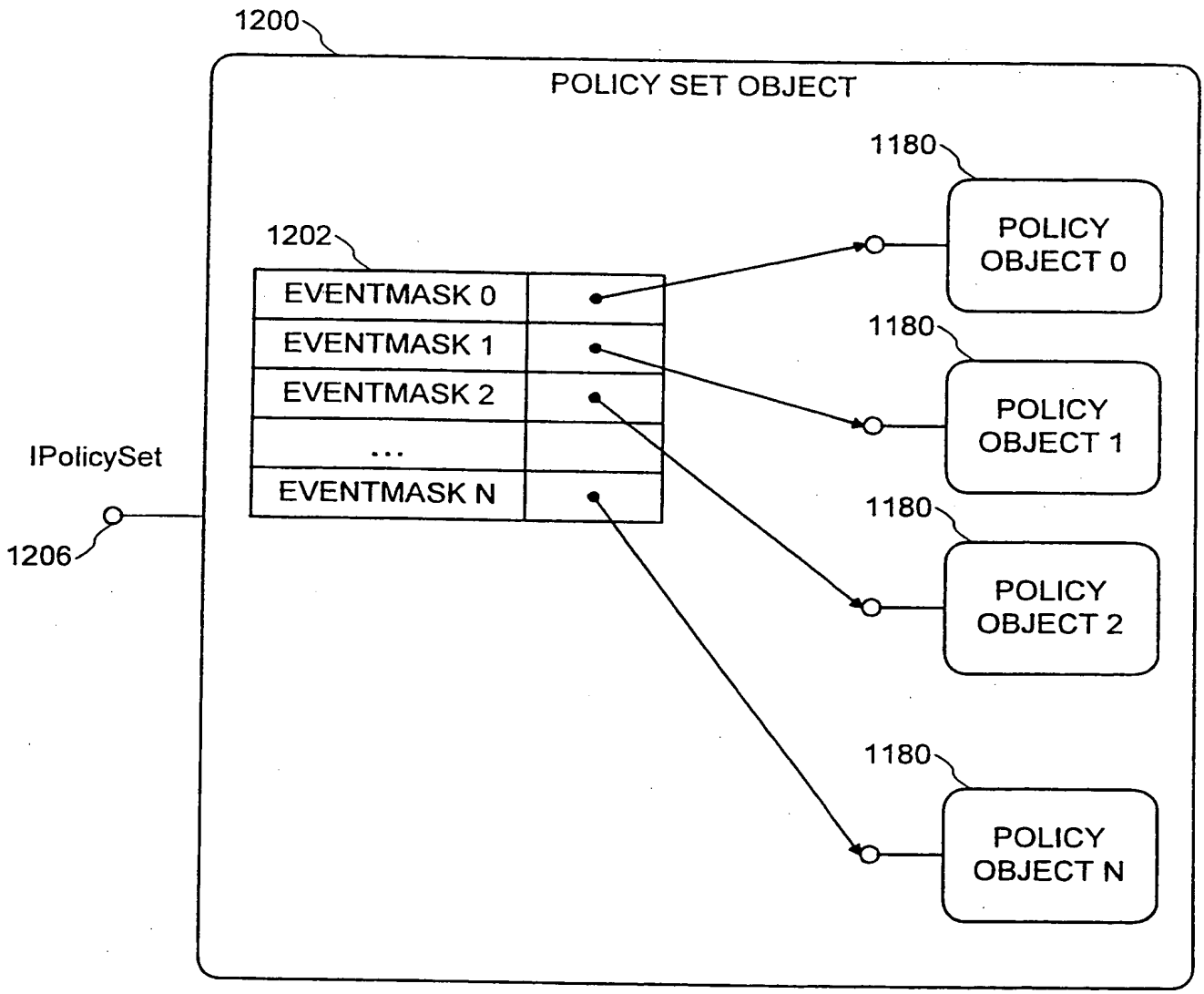


FIG. 18

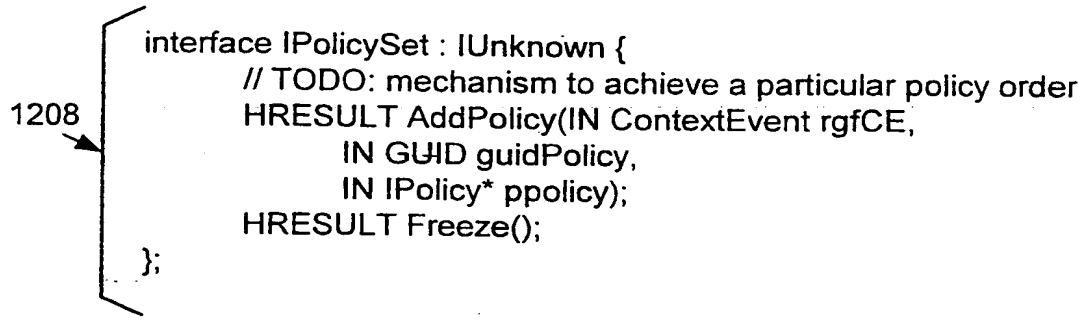


FIG. 19

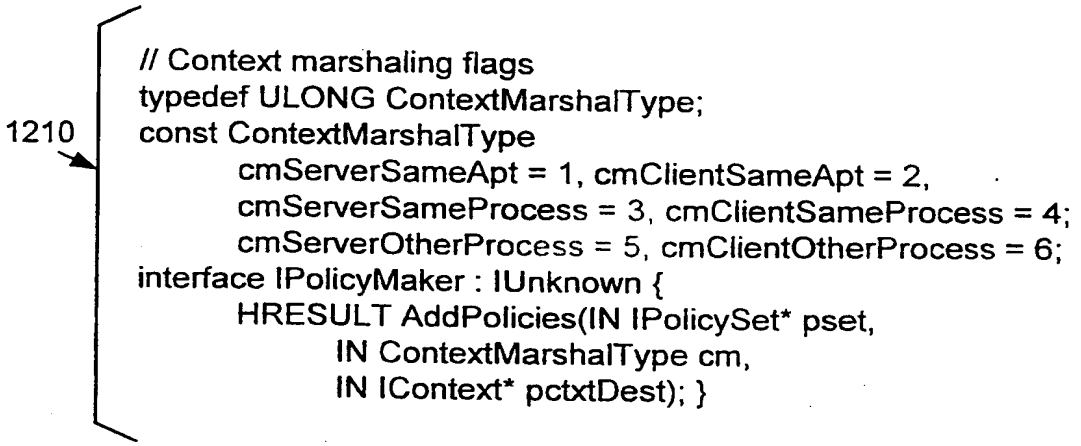


FIG. 20

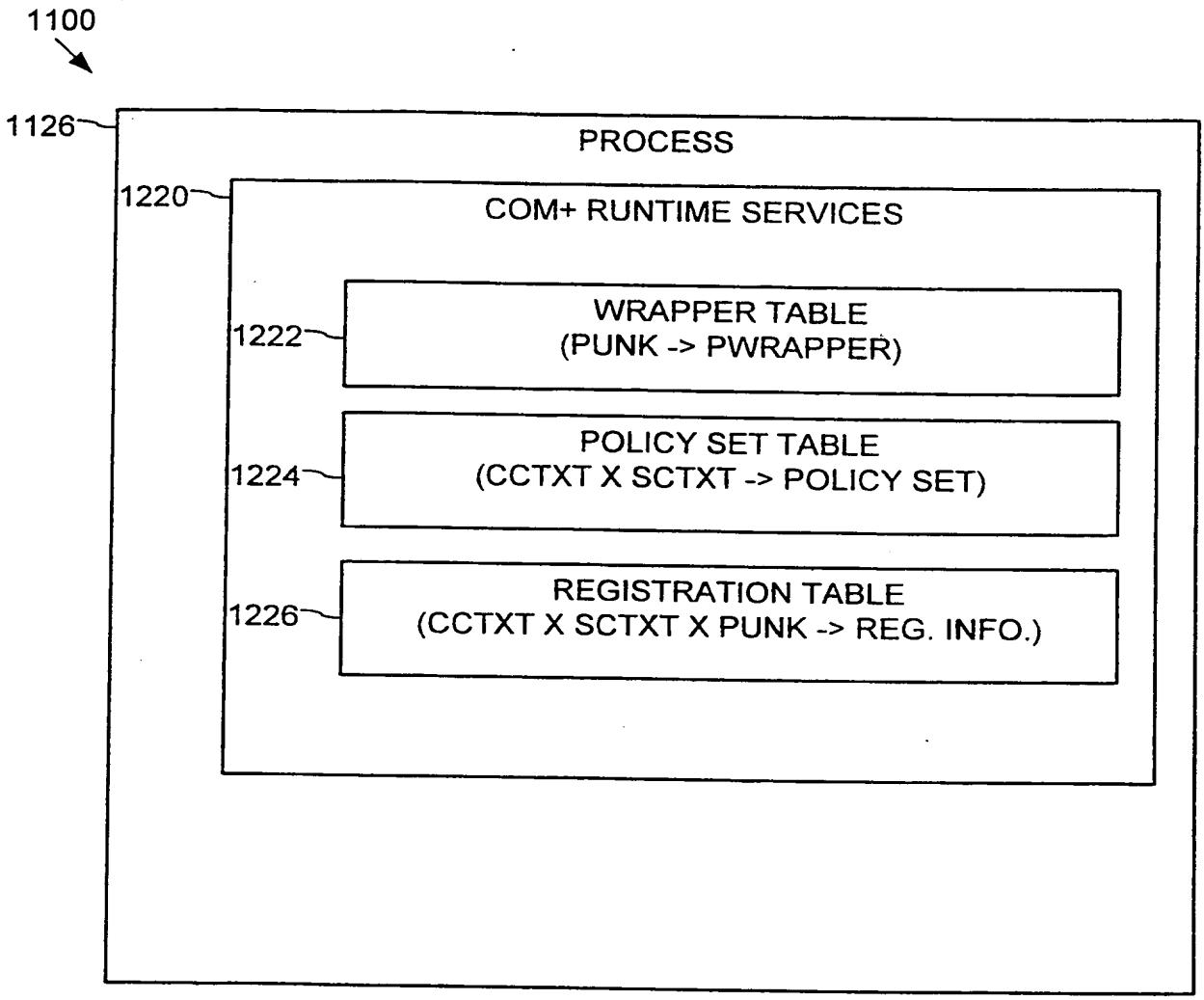


FIG. 21

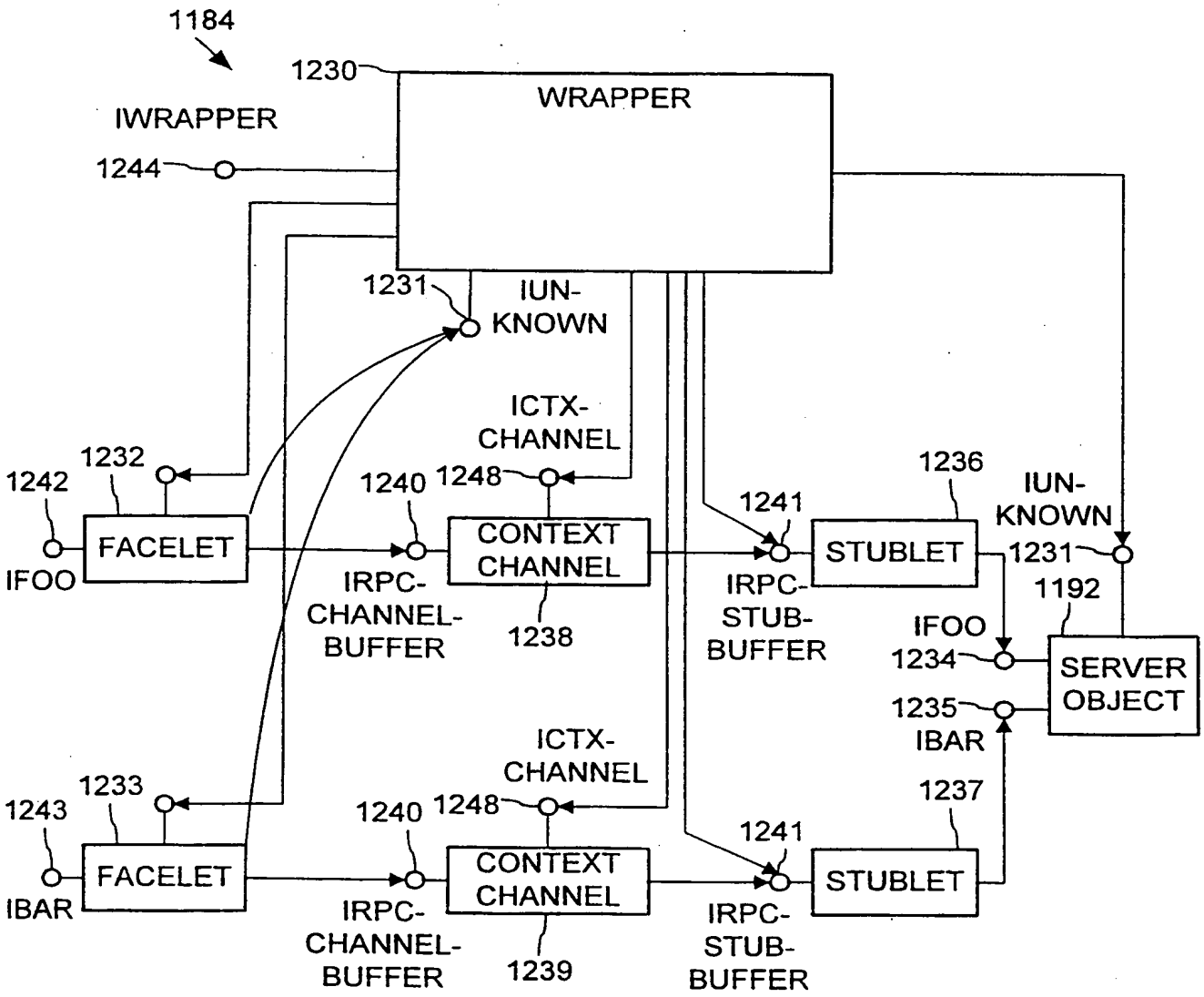


FIG. 22

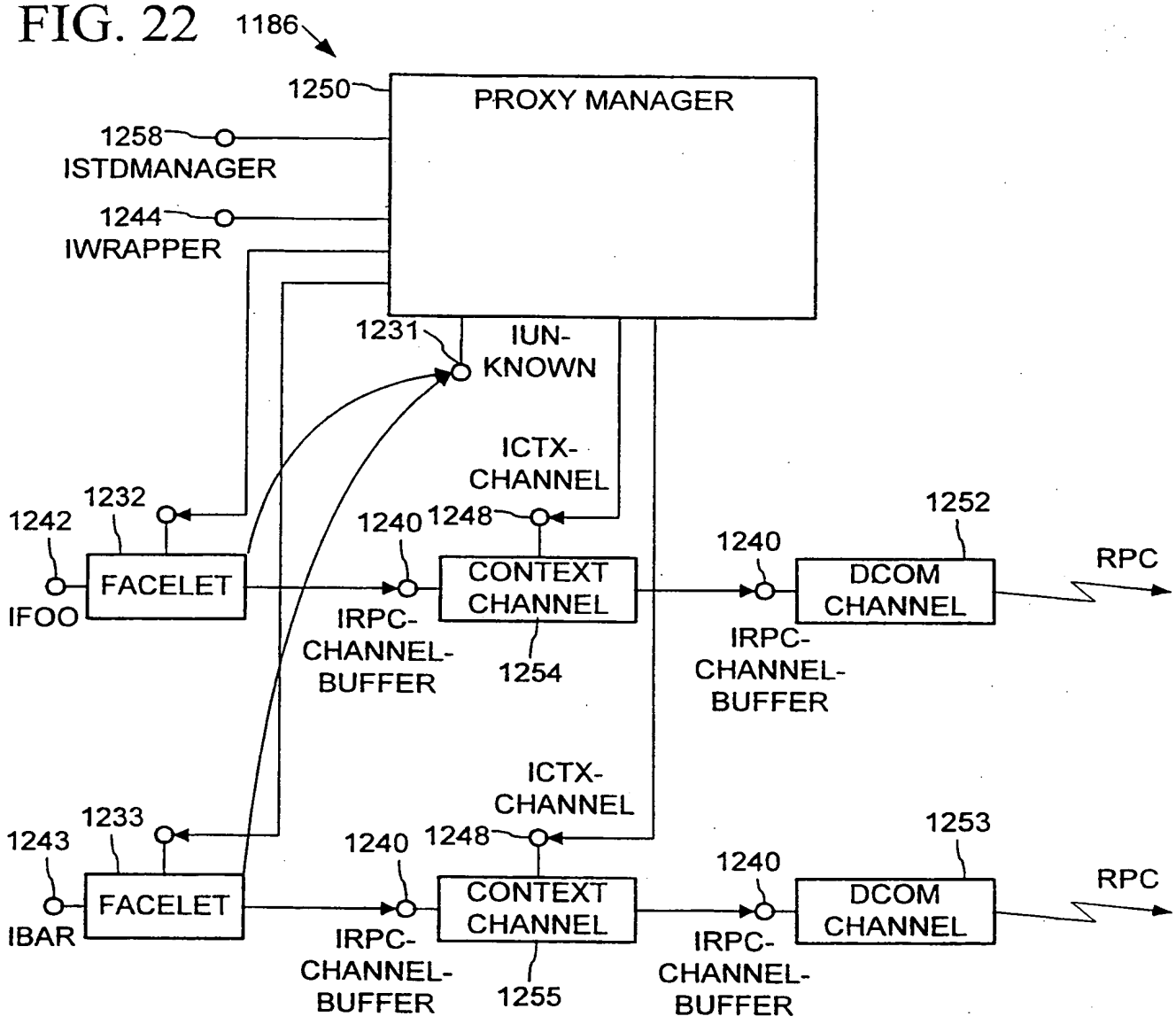


FIG. 23

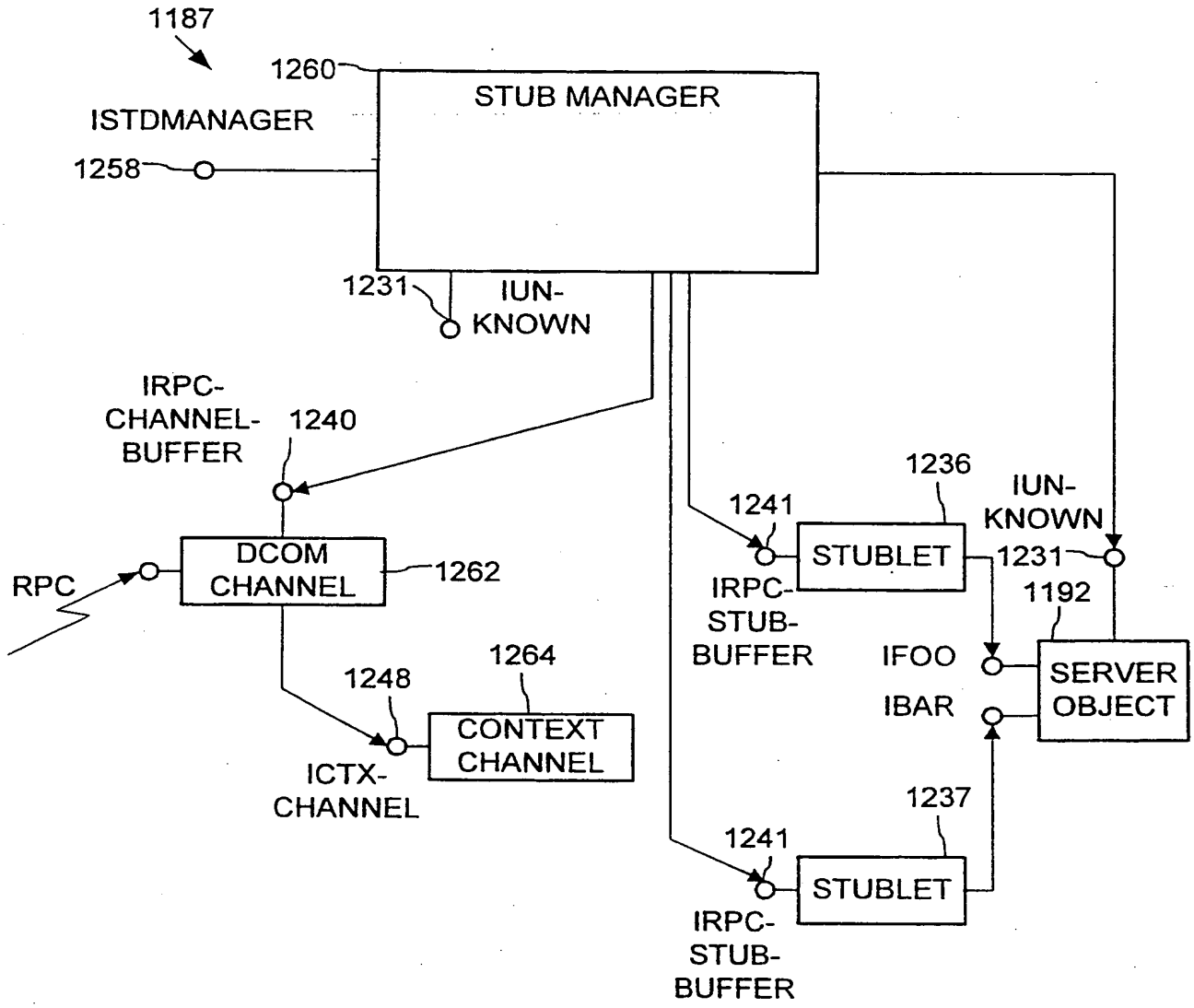


FIG. 24

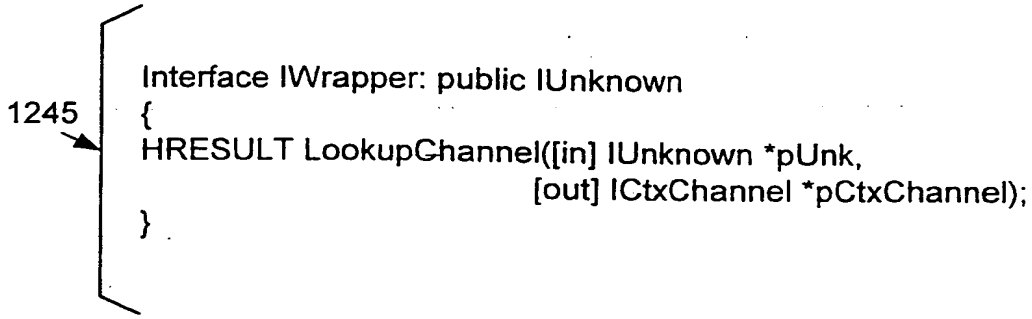


FIG. 25

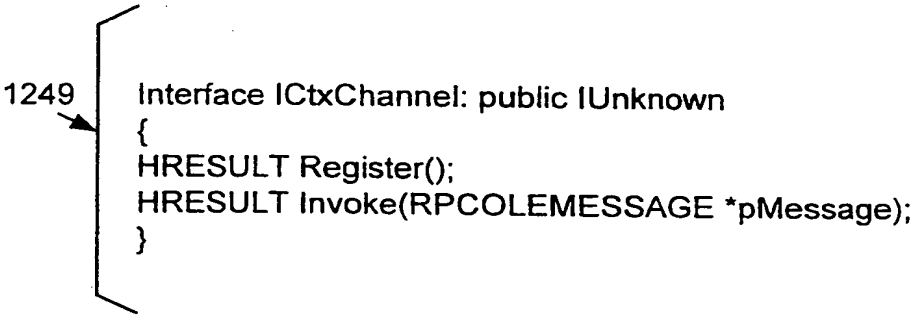


FIG. 26

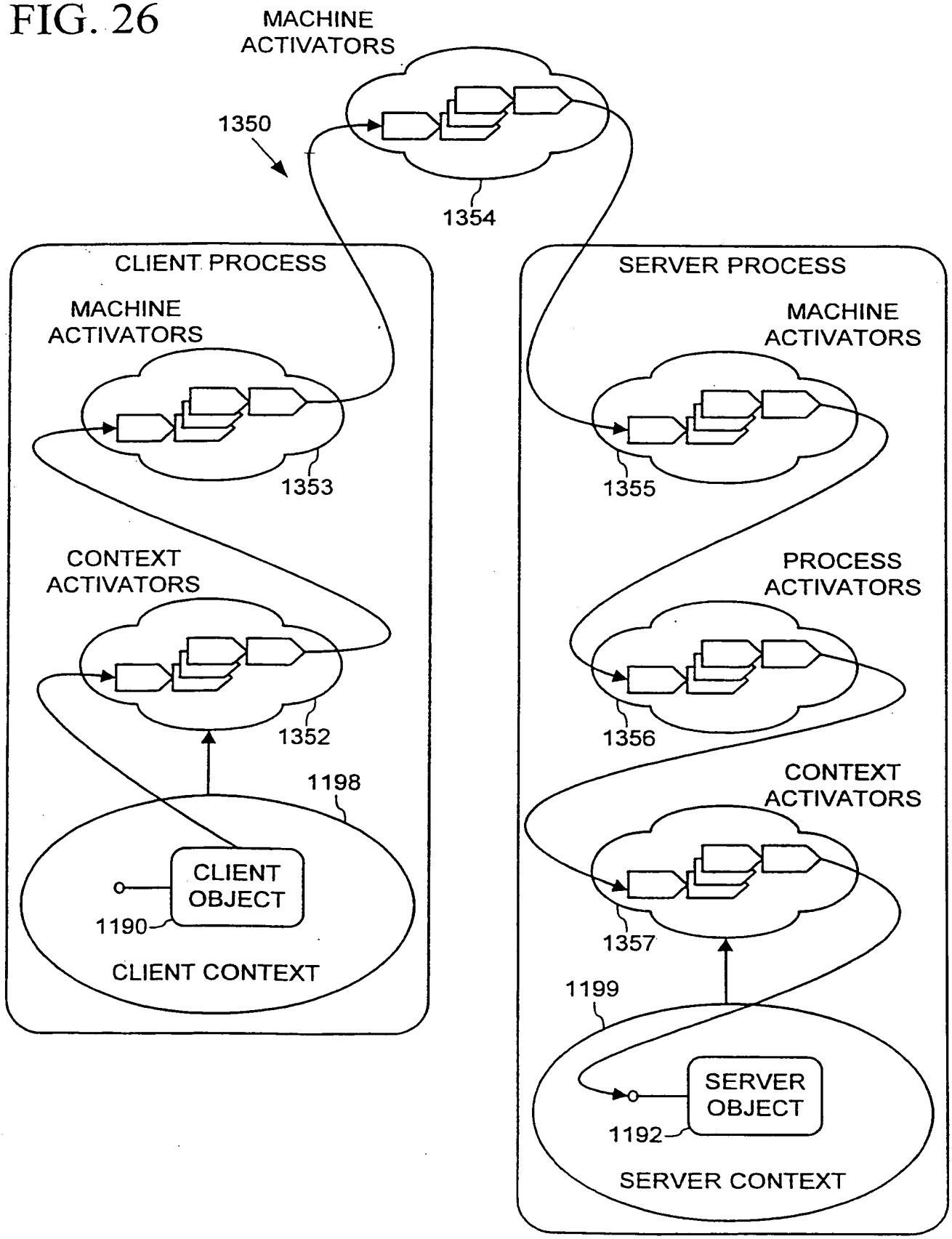


FIG. 27

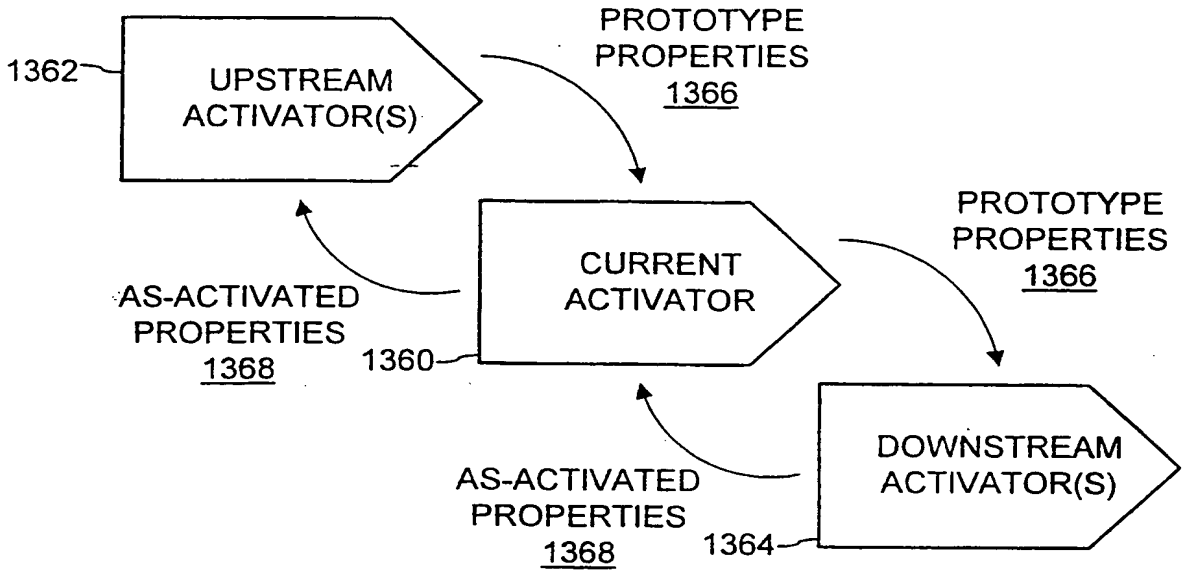


FIG. 28

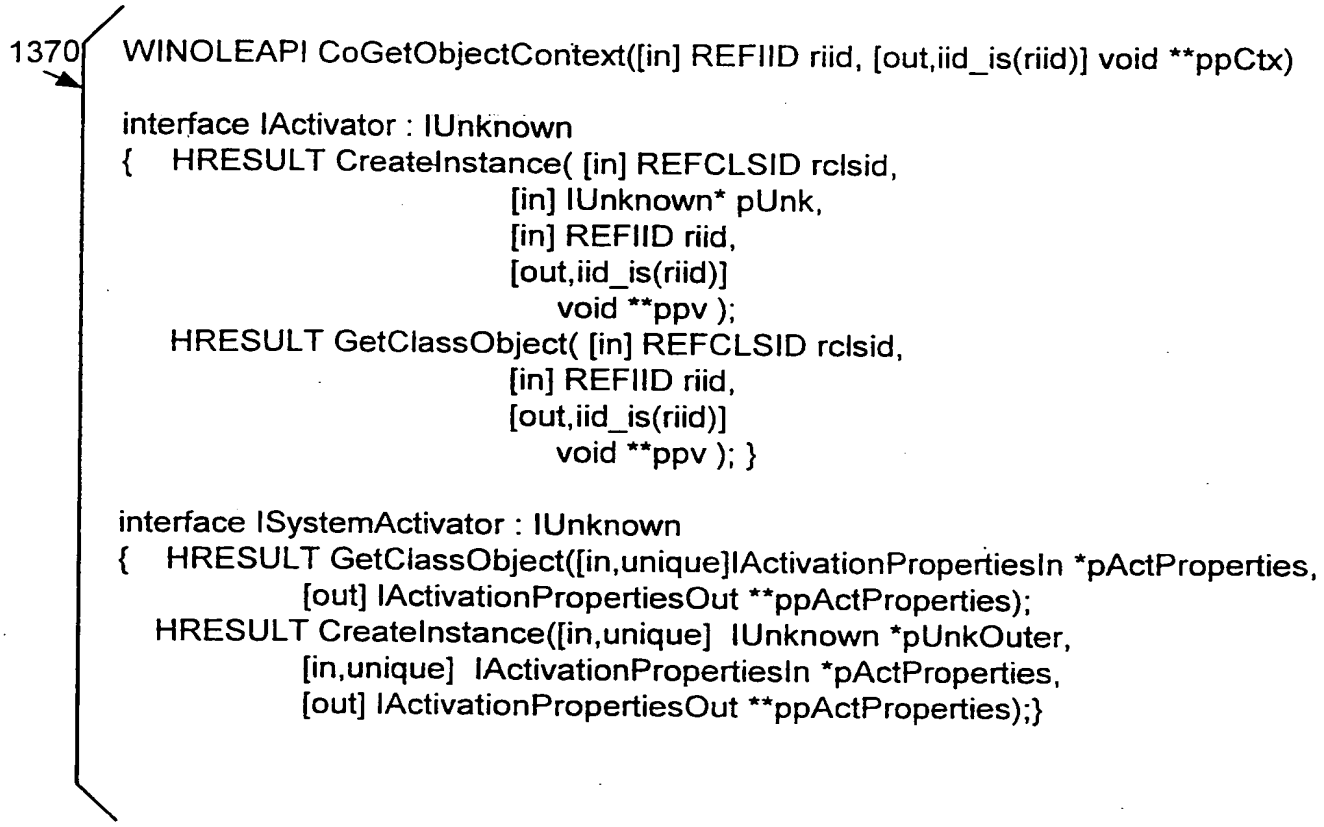


FIG. 29

1372

```
interface IInitActivationPropertiesIn : IUnknown
{   HRESULT SetClsctx ([in] DWORD clsctx);
    HRESULT SetClassInfo ([in,unique] IUnknown* pUnkClassInfo);
    HRESULT SetContextInfo ([in,unique] IContext* pClientContext,
                    [in] IContext* pPrototypeContext);
    HRESULT SetConstructFromStorage ([in,unique] IStorage* pStorage);
    HRESULT SetConstructFromFile ([in] WCHAR* wszFileName,
                    [in] DWORD dwMode);}

interface IActivationPropertiesIn : IUnknown
{   HRESULT GetActivationID([out] GUID *pActivationID);
    HRESULT GetClassInfo([in] REFIID riid, [out,iid_is(riid)] void** ppv);
    HRESULT GetClsctx([out] DWORD *pclsctx);
    HRESULT AddRequestedIIDs([in] DWORD clfs,
                    [in, size_is(clfs)] IID *rgIID);
    HRESULT GetRequestedIIDs ([out] ULONG* pulCount, [out] IID** prgIID);
    // The following two methods are for activators who delegate onward.
    HRESULT DelegateGetClassObject([out] IActivationPropertiesOut
**pActPropsOut);
    HRESULT DelegateCreateInstance([in] IUnknown *pUnkOuter,
                    [out] IActivationPropertiesOut **pActPropsOut);
    // The following method is for activators who do NOT delegate onward.
    HRESULT GetReturnActivationProperties([in] IUnknown *pUnk,
                    [out] IActivationPropertiesOut **ppActOut);}

interface IActivationPropertiesOut : IUnknown
{   HRESULT GetActivationID([out] GUID *pActivationID);
    HRESULT GetObjectInterface([in] REFIID riid,
                    [out, iid_is(riid)] void **ppv);
    HRESULT GetObjectInterfaces([in] DWORD clfs,
                    [in, size_is(clfs)] MULTI_QI *multiQi);}
```

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 9/46

A3

(11) International Publication Number: WO 00/10084

(43) International Publication Date: 24 February 2000 (24.02.00)

(21) International Application Number: PCT/US99/18813

(22) International Filing Date: 17 August 1999 (17.08.99)

(30) Priority Data:

09/135,106	17 August 1998 (17.08.98)	US
09/135,397	17 August 1998 (17.08.98)	US

(81) Designated States: JP, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(88) Date of publication of the international search report: 29 June 2000 (29.06.00)

(71) Applicant: MICROSOFT CORPORATION [US/US]; One Microsoft Way, Redmond, WA 98052 (US).

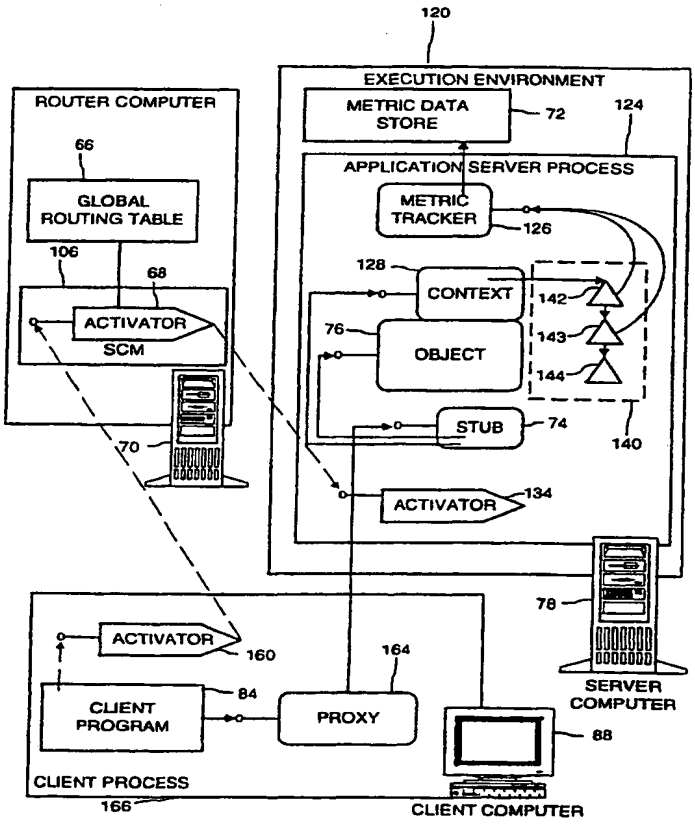
(72) Inventors: AL-GHOSEIN, Mohsen; 24677 S.E. 9th Place, Redmond, WA 98053 (US). GRAY, Jan, S.; 17214 N.E. 40th Street, Redmond, WA 98052 (US). MITAL, Amit; 13114 N.E. 111th Place, Kirkland, WA 98033 (US). LIMPRECHT, Rodney; 18806 N.E. 146th Way, Woodinville, WA 98072 (US).

(74) Agent: WIGHT, Stephen, A.; Klarquist, Sparkman, Campbell, Leigh & Whinston, LLP, One World Trade Center, Suite 1600, 121 S.W. Salmon Street, Portland, OR 97204 (US).

(54) Title: OBJECT LOAD BALANCING

(57) Abstract

A load balancing architecture routes object creation requests to server computers based on the object's class or some other characteristic. The architecture operates transparently to the object and the client program creating the object. A monitoring service at the servers observes a processing metric such as response time and provides the observations to a router computer. A load balancing engine at the router computer analyzes the observations to dynamically adjust routing, directing object creation requests to server computers having favorable performance. Analysis of observations is delayed to facilitate blending and dampening. Although a default load balancing engine is provided, software developers can specify a different engine per object class, and the engines can be tailored to the behavior of a particular object class. Load balancing is extensible to threads and processes. If performance at a server falls below an acceptable level, a rebalance policy transparently sends a rebalance message, breaking the connectionto the server.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

INTERNATIONAL SEARCH REPORT

Inter nal Application No
PCT/US 99/18813

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 459 837 A (CACCAVALE FRANK S) 17 October 1995 (1995-10-17) the whole document	1,13,20, 29,31, 34,38
A	EP 0 623 876 A (IBM) 9 November 1994 (1994-11-09) abstract column 1, line 1 -column 6, line 40 column 9, line 27 -column 11, line 31 column 19, line 18 -column 22, line 19 claims 1,6	1,13,20, 29,31, 34,38
A	EP 0 559 100 A (KITSUREGAWA MASARU ;MITSUBISHI ELECTRIC CORP (JP)) 8 September 1993 (1993-09-08) the whole document	1,13,20, 29,31, 34,38

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
2 March 2000	11 April 2000 (11.04.00)

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax (+31-70) 340-3016	Beltrán-Escavy, J

2

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inte ional Application No
PCT/US 99/18813

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	EP 0 638 863 A (NIPPON ELECTRIC CO) 15 February 1995 (1995-02-15) the whole document	1,13,20, 29,31, 34,38
A	US 5 442 791 A (GOODERUM MARK P ET AL) 15 August 1995 (1995-08-15) abstract column 1, line 1 -column 10, line 36 claims 1,11,15,18,28,33	1,13,20, 29,31, 34,38
Y	SCHIEMANN B ET AL: "A NEW APPROACH FOR LOAD BALANCING IN HIGH-PERFORMANCE DECISION SUPPORT SYSTEMS" FUTURE GENERATIONS COMPUTER SYSTEMS,NL,ELSEVIER SCIENCE PUBLISHERS. AMSTERDAM, vol. 12, no. 5, 1 April 1997 (1997-04-01), pages 345-355, XP000656288 ISSN: 0167-739X the whole document	36,37
Y	US 5 787 251 A (HAMILTON GRAHAM ET AL) 28 July 1998 (1998-07-28) abstract column 1, line 10 -column 1, line 16 column 4, line 10 -column 4, line 49 column 8, line 20 -column 18, line 59 claims 1,2	36,37

Form PCT/ISA/210 (continuation of second sheet) (July 1992)